

EXHIBIT 1

The X Window System

ROBERT W. SCHEIFLER

MIT Laboratory for Computer Science

and

JIM GETTYS

Digital Equipment Corporation and MIT Project Athena

An overview of the X Window System is presented, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. The system provides high-performance, high-level, device-independent graphics. A hierarchy of resizable, overlapping windows allows a wide variety of application and user interfaces to be built easily. Network-transparent access to the display provides an important degree of functional separation, without significantly affecting performance, which is crucial to building applications for a distributed environment. To a reasonable extent, desktop management can be custom-tailored to individual environments, without modifying the base system and typically without affecting applications.

Categories and Subject Descriptors: C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.4 [Computer-Communication Networks]: Distributed Systems—*distributed applications*; D.4.4 [Operating Systems]: Communication Management—*network communication*; *terminal management*; H.1.2 [Models and Principles]: User/Machine Systems—*human factors*; I.3.2 [Computer Graphics]: Graphics Systems—*distributed/network graphics*; I.3.4 [Computer Graphics]: Graphics Utilities—*graphics packages*; *software support*; I.3.6 [Computer Graphics]: Methodology and Techniques—*device independence*; *interaction techniques*

General Terms: Design, Experimentation, Human Factors, Standardization

Additional Key Words and Phrases: Virtual terminals, window managers, window systems

1. INTRODUCTION

The X Window System (or simply X) developed at MIT has achieved fairly widespread popularity recently, particularly in the UNIX¹ community. In this paper we present an overview of X, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. In X, this base window system provides high-performance graphics to a hierarchy of resizable windows. Rather than mandate a particular user interface, X provides primitives to support several policies and styles. Unlike most window systems, the base system in X is defined by a *network protocol*: asynchronous

¹UNIX is a trademark of AT&T Bell Laboratories.

Authors' addresses: R. W. Scheifler, 545 Technology Square, Cambridge, MA 02139; J. Gettys, Project Athena, MIT, Cambridge, MA 02139.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 0730-0301/86/0400-0079 \$00.75

stream-based interprocess communication replaces the traditional procedure call or kernel call interface. An application can utilize windows on any display in a network in a device-independent, network-transparent fashion. Interposing a network connection greatly enhances the utility of the window system, without significantly affecting performance. The performance of existing X implementations is comparable to that of contemporary window systems and, in general, is limited by display hardware rather than network communication. For example, 19,500 characters per second and 3500 short vectors per second are possible on Digital Equipment Corporation's VAXStation-II/GPX, both locally and over a local-area network, and these figures are very close to the limits of the display hardware.

X is the result of two separate groups at MIT having a simultaneous need for a window system. In the summer of 1984, the Argus system [16] at the Laboratory for Computer Science needed a debugging environment for multiprocess distributed applications, and a window system seemed the only viable solution. Project Athena [4] was faced with dozens, and eventually thousands, of workstations with bitmap displays and needed a window system to make the displays useful. Both groups were starting with the Digital VS100 display [14] and VAX hardware, but it was clear at the outset that other architectures and displays had to be supported. In particular, IBM workstations with bitmap displays of unknown type were expected eventually within Project Athena. Portability was therefore a goal from the start. Although all of the initial implementation work was for Berkeley UNIX, it was clear that the network protocol should not depend on aspects of the operating system.

The name X derives from the lineage of the system. At Stanford University, Paul Asente and Brian Reid had begun work on the W window system [3] as an alternative to VGTS [13, 22] for the V system [5]. Both VGTS and W allow network-transparent access to the display, using the synchronous V communication mechanism. Both systems provide "text" windows for ASCII terminal emulation. VGTS provides graphics windows driven by fairly high-level object definitions from a structured display file; W provides graphics windows based on a simple display-list mechanism, with limited functionality. We acquired a UNIX-based version of W for the VS100 (with synchronous communication over TCP [24] produced by Asente and Chris Kent at Digital's Western Research Laboratory. From just a few days of experimentation, it was clear that a network-transparent hierarchical window system was desirable, but that restricting the system to any fixed set of application-specific modes was completely inadequate. It was also clear that, although synchronous communication was perhaps acceptable in the V system (owing to very fast networking primitives), it was completely inadequate in most other operating environments. X is our "reaction" to W. The X window hierarchy comes directly from W, although numerous systems have been built with hierarchy in at least some form [11, 15, 18, 28, 30, 32-36]. The asynchronous communication protocol used in X is a significant improvement over the synchronous protocol used in W, but is very similar to that used in Andrew [10, 20]. X differs from all of these systems in the degree to which both graphics functions and "system" functions are pushed back (across the network) as application functions, and in the ability to tailor desktop management transparently.

The next section presents several high-level requirements that we believe a window system must satisfy to be a viable standard in a network environment, and indicates where the design of X fails to meet some of these requirements. In Section 3 we describe the overall X system model and the effect of network-based communication on that model. Section 4 describes the structure of windows, and the primitives for manipulating that structure. Section 5 explains the color model used in X, and Section 6 presents the text and graphics facilities. Section 7 discusses the issues of window exposure and refresh, and their resolution in X. Section 8 deals with input event handling. In Section 9 we describe the mechanisms for desktop management.

This paper describes the version² of X that is currently in widespread use. The design of this version is inadequate in several respects. With our experience to date, and encouraged by the number of universities and manufacturers taking a serious interest in X, we have designed a new version that should satisfy a significantly wider community. Section 10 discusses a number of problems with the current X design and gives a general idea of what changes are contemplated.

2. REQUIREMENTS

A window system contains many interfaces. A *programming* interface is a library of routines and types provided in a programming language for interacting with the window system. Both low-level (e.g., line drawing) and high-level (e.g., menus) interfaces are typically provided. An *application* interface is the mechanical interaction with the user and the visual appearance that is specific to the application. A *management* interface is the mechanical interaction with the user, dealing with overall control of the desktop and the input devices. The management interface defines how applications are arranged and rearranged on the screen, and how the user switches between applications; an individual application interface defines how information is presented and manipulated within that application. The *user* interface is the sum total of all application and management interfaces.

Besides applications, we distinguish three major components of a window system. The *window manager*³ implements the desktop portion of the management interface; it controls the size and placement of application windows, and also may control application window attributes, such as titles and borders. The *input manager* implements the remainder of the management interface; it controls which applications see input from which devices (e.g., keyboard and mouse). The *base window system* is the substrate on which applications, window managers, and input managers are built.

In this paper we are concerned with the base window system of X, with the facilities it provides to build applications and managers. The following requirements for the base window system crystallized during the design of X (a few were not formulated until late in the design process):

1. *The system should be implementable on a variety of displays.* The system should work with nearly any bitmap display and a variety of input devices. Our design focused on workstation-class display technology likely to be available in a

² Version 10.

³ Some people use this term for what we call the base window system; that is not the meaning here.

university environment over the next few years. At one end of the spectrum is a simple frame buffer and monochrome monitor, driven directly by the host CPU with no additional hardware support. At the other end of the spectrum is a multiplane display with color monitor, driven by a high-performance graphics coprocessor. Input devices, such as keyboards, mice, tablets, joysticks, light pens, and touch screens, should be supported.

2. *Applications must be device independent.* There are several aspects to device independence. Most important, it must not be necessary to rewrite, recompile, or even relink an application for each new hardware display. Nearly as important, every graphics function defined by the system should work on virtually every supported display; the alternative, which is to use GKS-style inquire operations [12] to determine the set of implemented functions at run time, leads to tedious case analysis in every application and to inconsistent user interfaces. A third aspect of device independence is that, as far as possible, applications should not need dual control paths to work on both monochrome and color displays.

3. *The system must be network transparent.* An application running on one machine must be able to utilize a display on some other machine, nor should it be necessary for the two machines to have the same architecture or operating system.

There are numerous examples of why this is important: a compute-intensive VLSI design program executing on a mainframe, but displaying results on a workstation; an application distributed over several stand-alone processors, but interacting with a user at a workstation; a professor running a program on one workstation, presenting results simultaneously on all student workstations.

In a network environment, there are certain to be applications that must run on particular machines or architectures. Examples include proprietary software, applications depending on specific architectural properties, and programs manipulating large databases. Such applications still should be accessible to all users. In a truly heterogeneous environment, not all programming languages and programming systems are supported on all machines, and it is very undesirable to have to write an interactive front end in multiple languages in order to make the application generally available. With network-transparent access, this is not necessary; a single front end written in the same language as the application suffices.

One might think that remote display will be extremely infrequent, and that performance is therefore much less important than for local display. Experience at MIT, however, indicates that many users routinely make use of the remote display capabilities in X, and that the performance of remote display is quite important. The desktop display, although physically connected to a single computer, is used as a true *network virtual terminal*; indeed, the idea of an X server (see the next section) built into a Blit-like terminal [23] is an intriguing one.

4. *The system must support multiple applications displaying concurrently.* For example, it should be possible to display a clock with a sweep second hand in one window, while simultaneously editing a file in another window.

5. *The system should be capable of supporting many different application and management interfaces.* No single user interface is “best”; different communities have radically different ideas about user interfaces. Even within a single community, “experts” and “novices” place different demands on an interface. Instead of mandating a particular user interface, the base window system should support a wide range of interfaces.

To achieve this, the system must provide *hooks* (mechanism) rather than *religion* (policy). For example, since menu styles and semantics vary dramatically among different user interfaces, the base window system must provide primitives from which menus can be built, instead of just providing a fixed menu facility.

The system should be designed in such a way that it is possible to implement management policy in a way that is external to the base window system and external to applications. Applications should be largely independent of management policy and mechanism; applications should *react* to management decisions, rather than *direct* those decisions. For example, an application needs to be informed when one of its windows is resized and should react by reformatting the information displayed, but involvement of the application should not be required in order for the user to change the size. Making applications management independent, as well as device independent, facilitates the sharing of applications among diverse cultures.

6. *The system must support overlapping windows, including output to partially obscured windows.* This is in some sense a by-product of the previous requirement, but it is important enough to merit explicit statement. Not all user interfaces allow windows to overlap arbitrarily. However, even interfaces that do not allow application windows to overlap typically provide some form of pop-up menu that overlaps application windows. If such menus are built from windows, then support for overlapping windows must exist.

7. *The system should support a hierarchy of resizable windows, and an application should be able to use many windows at once.* Subwindows provide a clean, powerful mechanism for exporting much of the basic system machinery back to the application for direct use. Many applications make use of their own window-like abstractions; some even implement what is essentially another window system, nested within the “real” window system. It is important to support arbitrary levels of nesting. What is viewed as a single window at one abstraction level may well require multiple subwindows at a lower level. By providing a true window hierarchy, application windows can be implemented as true windows within the system, freeing the application from duplicating machinery such as clipping and input control.

8. *The system should provide high-performance, high-quality support for text, 2-D synthetic graphics, and imaging.* The base window system must provide “immediate” or “transparent” graphics: The application describes the image precisely, and the system does not attempt to second-guess the application. The use of high-level models, whereby the application describes *what* it wants in terms of fairly abstract objects and the system determines *how* best to render the

image, cannot be imposed as the only form of graphics interface. Such models generally fail to provide adequate support for some important class of applications, and different user communities tend to have strong opinions about which model is “best.” It is extremely important to provide high-level models, but they should be built in layers on top of the base window system.

Support for 3-D graphics is not listed as a requirement, but this is not to say it is unimportant. We simply have not considered 3-D graphics, owing to lack of expertise and lack of time.

9. *The system should be extensible.* For example, the core system may not support 3-D graphics, but it should be possible to extend the system with such support. The extension mechanism should allow communities to extend the system noncooperatively, yet allow such independent extensions to be merged gracefully.

We believe that a window system must satisfy these requirements to be a viable standard in an environment of high-performance workstations and mainframes connected via high-performance local-area networks. X satisfies most of these requirements, but currently fails to satisfy a few owing to practical considerations of staffing and time constraints: The design and much of the implementation of the base window system were to be handled solely by the first author; it was important to get a working system up fairly quickly; and the immediate applications only required relatively simple text and graphics support. As a result, X is not designed to handle high-end color displays or to deal with input devices other than a keyboard and mouse, some support for high-quality text and graphics is missing, X only provides support for one class of management policy, and no provision has been made for extensions. As discussed in Section 10, these and other problems are being addressed in a redesign of X.

3. SYSTEM MODEL

The X window system is based on a client-server model (see Figure 1); this model follows naturally from requirements 2 and 3 in the previous section. For each physical display, there is a controlling server. A client application and a server communicate over a reliable duplex (8-bit) byte stream. A simple block-stream protocol is layered on top of the byte stream. If the client and server are on the same machine, the stream is typically based on a local interprocess communication (IPC) mechanism; otherwise a network connection is established between the pair. Requiring nothing more than a reliable duplex byte stream (without urgent data) for communication makes X usable in many environments. For example, the X protocol can be used over TCP [24], DECnet [38], and Chaos [19].

Multiple clients can have connections open to a server simultaneously, and a client can have connections open to multiple servers simultaneously. The essential tasks of the server are to multiplex requests from clients to the display, and demultiplex keyboard and mouse input back to the appropriate clients. Typically, the server is implemented as a single sequential process, using round-robin scheduling among the clients, and this centralized control trivially solves many

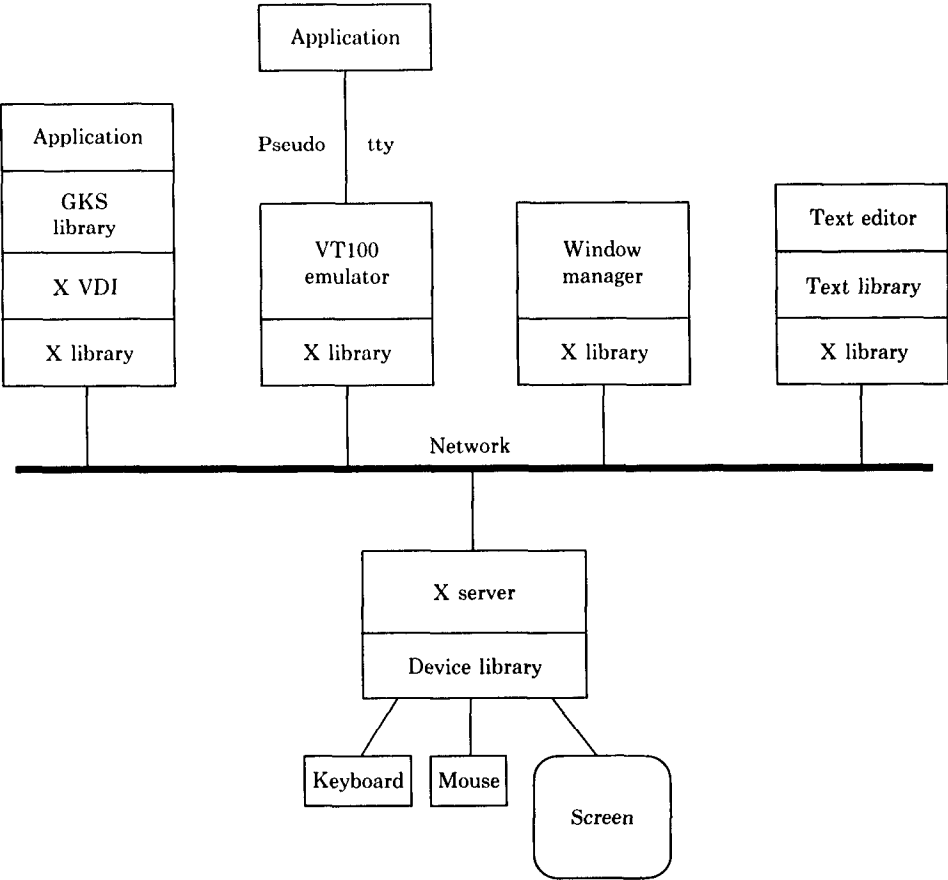


Fig. 1. System structure.

synchronization problems; however, a multiprocess server has also been implemented. Although one might place the server in the kernel of the operating system in an attempt to increase performance, a user-level server process is vastly easier to debug and maintain, and performance under UNIX in fact does not seem to suffer. Similar performance results have been obtained in Andrew [10]. Various tricks are used in both clients and server to optimize performance, principally by minimizing the number of operating system calls [9].

The server encapsulates the base window system. It provides the fundamental resources and mechanisms, and the hooks required to implement various user interfaces. All device dependencies are encapsulated by the server; the communication protocol between clients and server is device independent. By placing all device dependencies on one end of a network connection, applications are truly device independent. The addition of a new display type simply requires the addition of a new server implementation; no application changes are required. Of course, the server itself is designed as device-independent code layered on

top of a device-dependent core, so only the “back end” of the server need be reimplemented for each new display.⁴

3.1 Network Considerations

It is extremely important for the server to be robust with respect to client failures. The server and the network protocol must be designed so that the server never trusts clients to provide correct data. As a corollary, the protocol must be designed in such a way that, if the server ever has to wait for a response from a client, it must be possible to continue servicing other clients. Without this property a buggy client or a network failure could easily cause the entire display to freeze up.

Byte ordering [6] is a standard problem in network communication: When a 16- or 32-bit quantity is transmitted over an 8-bit byte stream, is the most significant byte transmitted first (big-endian byte order) or is the least significant byte transmitted first (little-endian byte order)? Some machines with byte-addressable memory use big-endian order internally, and others use little-endian order. If a single order is chosen for network communication, some machines will suffer the overhead of swapping bytes, even when communicating with a machine using the same internal byte order. Such an approach also means that both parties in the communication must worry about byte order.

The X protocol uses a different approach. The server is designed to accept both big-endian and little-endian connections. For example, using TCP this is accomplished by having the server listen on two distinct ports; little-endian clients connect to the server on one port, and big-endian clients connect on the other. Clients always transmit and receive in their native byte order. The server alone is responsible for byte swapping, and byte swapping only occurs between dissimilar architectures. This eliminates the byte swapping overhead in the most common situations and greatly simplifies the building of client-side interface libraries in various programming languages. X is not unique in its use of this trick; the current VGTS implementation uses the same trick, and similar protocol optimizations have been used in various network-based applications.

Another potential problem in protocol design is word alignment. In particular, some architectures require 16-bit quantities to be aligned on 16-bit boundaries and 32-bit quantities to be aligned on 32-bit boundaries in memory. To allow efficient implementations of the protocol across a spectrum of 16- and 32-bit architectures, the protocol is defined to consist of blocks that are always multiples of 32 bits, and each 16- and 32-bit quantity within a block is aligned on 16- and 32-bit boundaries, respectively.

X is designed to operate in an environment where the interprocess communication round-trip time is between 5 and 50 milliseconds (ms), both for local and for network communication. We also assume that data transmission rates are comparable to display rates; for example, to transmit and display 5000 characters per second, a data rate of approximately 50 kilobits per second (kbits/s) will be needed, and to transmit and display 20,000 characters per second, a data rate of

⁴ A back end has been implemented using a programming interface to X itself, such that a complete “recursive” X server executes inside a window of another X server.

approximately 200 kbits/s will be needed. Networks and protocol implementations with these characteristics are now quite commonplace. For example, workstations running Berkeley UNIX, connected via 10-megabit-per-second (Mbit/s) local area networks, typically have round-trip times of 15 to 30 ms, and data rates of 500 kbits/s to 1 Mbit/s.

The round-trip time is important in determining the form of the communication protocol. Text and graphics are the most common requests sent from a client to the server; examples of individual requests might be to draw a string of text or to draw a line. Such requests could be sent either synchronously, in which case the client sends a request only after receiving a reply from the server to the previous request, or they could be sent asynchronously, without the server generating any replies. However, since the requests are sent over a reliable stream, they are guaranteed to arrive and arrive in order, so replies from the server to graphics requests serve no useful purpose. Moreover, with round-trip times over 5 ms, output to the display must be asynchronous, or it will be impossible to drive high-speed displays adequately. For example, at 80 characters per request and a 25-ms round-trip time, only 3200 characters per second can be drawn synchronously, whereas many hardware devices are capable of displaying between 5000 and 30,000 characters per second.

Similarly, polling the server for keyboard and mouse input would be unacceptable in many applications, particularly those written in sequential languages. For example, an application attempting to provide real-time response to input has to poll periodically for input during screen updates. For an application with a single thread of control, this effectively results in synchronous output and consequent performance loss. Hence, input must be generated asynchronously by the server, so that applications need at most perform local polling.

The round-trip time is also important in determining what user interfaces can be supported without embedding them directly in the server. The most important concern is whether remote, application-level mouse tracking is feasible. By *tracking*, we do not mean maintaining the cursor image on the screen as the user moves the mouse; that function is performed autonomously by the X server, often directly in hardware. Rather, applications track the mouse by animating some other image on the screen in real time as the mouse moves. For round-trip times under 50 ms, tracking is perfectly reasonable, driven either by motion events generated by the server or by continuous polling from the application. With a refresh occurring up to 30 times every second, remote tracking is demonstrably “instantaneous” with mouse motion.

For tracking to be effective, however, relatively little time can be spent updating the display at each movement, so typically only relatively small changes can be made to the screen while tracking. This is certainly the case for simple tracking, such as rubber banding window outlines and highlighting menu items. It might be argued that the ability to run application-specific code in the server is required for acceptable hand-eye coordination during more complex tracking. For example, NeWS [31] provides such a mechanism in a novel way. However, we are not convinced there are sufficient benefits to justify such complexity. Typically, complex tracking is bound intimately to application-specific data structures and knowledge representations. The information needed by the front end for tracking

is intertwined with the information needed by the back end for the “real” work; the information cannot be reasonably separated or duplicated. It is simply unreasonable to believe that applications will download large, complex front ends into a server; communication round-trip times are a reality that cannot be escaped.

3.2 Resources

The basic resources provided by the server are windows, fonts, mouse cursors, and offscreen images; later sections describe each of these. Clients request creation of a resource by supplying appropriate parameters (such as the name of the font); the server allocates the resource and returns a 29-bit⁵ unique identifier used to represent it. The use and interpretation of a resource identifier are independent of any network connection. Any client that knows (or guesses) the identifier for a resource can use and manipulate the resource freely, even if it was created by another client. This capability is required to allow window managers to be written independently of applications, and to allow multiprocess applications to manipulate shared resources. However, to avoid problems associated with clients that fail to clean up their resources at termination (which is all too common in operating systems where users can unilaterally abort processes), the maximum lifetime of a resource is always tied to the connection over which it was created. Thus, when a client terminates, all of the resources it created are destroyed automatically.

Access control is performed only when a client attempts to establish a connection to the server; once the connection is established, the client can freely manipulate any resource. Since accidental manipulation of some other client's resource is extremely unlikely (both in theory and in practice), we believe introducing access control on a per-resource basis would only serve to decrease performance, not to significantly increase security or robustness. The current access control mechanism is based simply on host network addresses, as this information is provided by most network stream protocols, and there seems to be no widely used or even widely available user-level authentication mechanism. Host-based access control has proved to be marginally acceptable in a workstation environment, but is rather unacceptable for time-shared machines.⁶

Each client-generated protocol request is a simple data block consisting of an opcode, some number of fixed-length parameters, and possibly a variable-length parameter. For example, to display text in a window, the fixed-length parameters include the drawing color and the identifiers for the window and the font, and the variable-length parameter is the string of characters. All operations on a resource explicitly contain the identifier of the resource as a parameter. In this way, an application can multiplex use of many windows over a single network connection. This multiplexing makes it easy for the client to control the time order of updates to multiple windows; if a separate stream was used for each window, time order could not be controlled without strong guarantees from the stream mechanism. Similarly, each input event generated by the server contains

⁵ To simplify implementation in languages built with garbage collection, high-order bits are not used.

⁶ It is interesting that *professors* at MIT have argued vociferously to disable all access control.

the identifier of the window in which the event occurred. Multiplexing over a single stream allows the client to act on events from multiple windows in correct time order; again, the use of a stream per window would not allow such ordering, even if the events carry timestamps.

Numerous UNIX-based window systems [17, 20, 21, 30, 36] use file or channel descriptors to represent windows; window creation involves an interaction with the operating system, which results in the creation of such a descriptor. Typically, this means the window cannot be named (and hence cannot be shared) by programs running on different machines, and perhaps not even by programs running on the same machine. More serious, there is often a severe restriction on the number of active descriptors a process may have: 20 on older systems and usually 64 on newer systems. The use of 50 or more windows (albeit nested inside a single top-level window) is quite common in X applications. The use of a single connection, over which an arbitrary number of windows can be multiplexed, is clearly a better approach.

4. WINDOW HIERARCHY

The server supports an arbitrarily branching hierarchy of rectangular windows. At the top is the *root* window, which covers the entire screen. The *top-level* windows of applications are created as subwindows of the root window. The window hierarchy models the now-familiar “stacks of papers” desktop. For a given window, its subwindows can be stacked in any order, with arbitrary overlaps. When window W1 partially or completely covers window W2, we say that W1 *obscures* W2. This relationship is not restricted to siblings; if W1 obscures W2, then W1 may also obscure subwindows of W2. A window also obscures its parent. Window hierarchies never interleave; if window W1 obscures sibling window W2, then subwindows of W2 never obscure W1 or subwindows of W1. A window is not restricted in size or placement by the boundaries of its parent, but a window is always visibly clipped by its parent: Portions of the window that extend outside the boundaries of the parent are never displayed and do not obscure other windows. Finally, a window can be either *mapped* or *unmapped*. An unmapped window is never visible on the screen; a mapped window can only be visible if all of its ancestors are also mapped.

Output to a leaf window (one with no subwindows) is always clipped to the visible portions of the window; drawing on such a window never draws into obscuring windows. Output to a window that contains subwindows can be performed in two modes. In *clipped* mode the output is clipped normally by all obscuring windows (including subwindows), but in *draw-through* mode the output is not clipped by subwindows. For example, draw-through mode is used on the root window during window management, tracking the mouse with the outline of a window to indicate how the window is to be moved or resized. If clipped mode were used instead, the entire outline would not be visible.

The coordinate system is defined with the X axis horizontal and the Y axis vertical. Each window has its own coordinate system, with the origin at the upper left corner of the window. Having per-window coordinate systems is crucial, particularly for top-level windows; applications are almost always designed to be insensitive to their position on the screen, and having to worry about race

conditions when moving windows would be a disaster. The coordinate system is discrete: Each pixel in the window corresponds to a single unit in the coordinate system, with coordinates centered on the pixels, and all coordinates are expressed as integers in the protocol. We believe fractional coordinates are not required at the protocol level for the raster graphics provided in X (see Section 6), although they may be required for high-end color graphics, such as antialiasing. The aspect ratio of the screen is not masked by the protocol, since we believe that most displays have a one-to-one aspect ratio; in this regard X is arguably device dependent.

Although the coordinate system is discrete at the protocol level, continuous or alternate-origin coordinate systems certainly can be used at the application level, but client-side libraries must eventually translate to the discrete coordinates defined by the protocol. In this way, we can ignore the many variations in floating-point (or even fixed-point) formats among architectures. Further, the coordinates can be expressed in the protocol as 16-bit quantities, which can be manipulated efficiently in virtually every machine/display architecture and which minimizes the number of data bytes transmitted over the network. The use of 16-bit quantities does have a drawback, in that some applications (particularly CAD tools) like to perform zoom operations simply by scaling coordinates and redrawing, relying on the window system to clip appropriately. Since scaling quickly overflows 16 bits, additional clipping must be performed explicitly by such applications.

A window can optionally have a *border*, a shaded outer frame maintained explicitly by the X server. The origin of the window's coordinate system is inside the border, and output to the window is clipped automatically so as not to extend into the border. The presence of borders slightly complicates the semantics of the window system; for simplicity we ignore them in the remainder of this paper.

The basic operations on window structure are straightforward. An unmapped window is created by specifying the parent window, the position within the parent of the upper left corner of the new window, and the width and height (in coordinate units) of the new window. A window can be destroyed, in which case all windows below it in the hierarchy are also destroyed. A window can be mapped and unmapped, without changing its position. A window can be moved and resized, including being moved and resized simultaneously. A window can also be "depthwise" raised to the top or lowered to the bottom of the stack with respect to its siblings, without changing its coordinate position. Currently mapping or configuring a window forces the window to be raised. This restriction appeared to simplify the server implementation but also happened to match the basic management interface we expected to build. This restriction will be eliminated in the next version.

The windows described above are the usual *opaque* windows. X also provides *transparent* windows. A transparent window is always invisible on the screen and does not obscure output to, or visibility of, other windows. Output to a transparent window is clipped to that window but is actually drawn on the parent window. Thus, for output, a transparent window is simply a clipping rectangle that can be applied to restrict output within a (parent) window. Input processing for transparent and opaque windows is identical, as described in Section 8. In

Section 10 we argue that most uses of transparent windows are better satisfied with other mechanisms. Therefore, for simplicity, we ignore transparent windows in the rest of this paper.

The X server is designed explicitly to make windows inexpensive. Our goal was to make it reasonable to use windows for such things as individual menu items, buttons, and even individual items in forms and spreadsheets. As such, the server must deal efficiently with hundreds (though not necessarily thousands) of windows on the screen simultaneously. Experience with X has shown that many implementors find this capability extremely useful, particularly when building extensible tool kits.

5. COLOR

The screen is viewed as two dimensional, with an N -bit *pixel* value stored at each coordinate. The number of bits in a pixel value and how a value translates into a color depend on the hardware. X is designed to support two types of hardware: monochrome and pseudocolor. A monochrome display has 1 bit per pixel, and the two values translate into black and white. Pseudocolor displays typically have between 4 and 12 bits per pixel; the pixel value is used as an index into a color map, yielding red, green, and blue intensities. The color map can be changed dynamically, so that a given pixel value can represent different colors over time. Gray scale is viewed as a degenerate case of pseudocolor.

We desire a design matching most display hardware, while abstracting differences in such a way that programmers do not have to double- or triple-code their applications to cover the spectrum. We also want multiple applications to coexist within a single color map, so that applications always show true color on the screen. To allow this and to keep applications device independent, pixel values should not be coded explicitly into applications. Instead, the server must be responsible for managing the color map, and color map allocation must be expressed in hardware-independent terms.

All graphics operations in X are expressed in terms of pixel values. For example, to draw a line, one specifies not only the coordinates of the endpoints, but the pixel value with which to draw the line. (Logic functions and plane-select masks are also specified, as described in Section 6.) On a monochrome display, the only two pixel values are 0 and 1, which are (somewhat arbitrarily) defined to be black and white, respectively. On a pseudocolor display, pixel values 0 and 1 are preallocated by the server for use as “black” and “white” so that monochrome applications display correctly on color displays. Of course, the actual colors need not be black and white, but can be set by the user.

There are two ways for a client to obtain pixel values. In the simplest request the client specifies red, green, and blue color values, and the server allocates an arbitrary pixel value and sets the color map so that the pixel value represents the closest color the hardware can provide. The color map entry for this pixel value cannot be changed by the client, so, if some other client requests an equivalent color, the server is free to respond with the same pixel value. Such sharing is important in maximizing use of the color map. To isolate applications from variations in color representation among displays (e.g., due to the standard of illumination used for calibration), the server provides a color database that

clients can use to translate string names of colors into red, green, and blue values tailored for the particular display.

The second request allocates writable map entries. This mechanism was designed explicitly for X; we are not aware of a comparable mechanism in any other window system. The client specifies two numbers, C and P , with C positive and P nonnegative; the request can be expressed as “allocate C colors and P planes.” The total number of pixel values allocated by the server is $C \times 2^P$. The values passed back to the client consist of C base pixel values and a plane mask containing P bits. None of the base pixel values have any 1 bits in common with the plane mask, and the complete set of allocated pixel values is obtained by combining all possible combinations of 1 bits from the plane mask with each of the base pixel values. The client can optionally require the P planes to be contiguous, in which case all P bits in the plane mask will be contiguous.

There are three common uses of this second request. One is simply to allocate a number of “unrelated” pixel values; in this case P will be 0. A second use is in imaging applications, where it is convenient to be able to perform simple arithmetic on pixel values. In this case a contiguous block of pixel values is allocated by setting C to 1 and P to the log (base 2) of the number of pixel values required, and requesting contiguous allocation. Arithmetic on the pixel values then requires at most some additional shift and mask operations.

A third form of allocation arises in applications that want some form of overlay graphics, such as highlighting or outlining regions. Here the requirement is to be able to draw and then erase graphics without disturbing existing window contents. For example, suppose an application typically uses four colors, but needs to be able to overlay a rectangle outline in a fifth color. An allocation request with C set to 4 and P set to 1 results in two groups of four pixel values. The four base pixel values are assigned the four normal colors, and the four alternate pixel values are all assigned the fifth color. Overlay graphics can then be drawn by restricting output (see the next section) to the single bit plane specified in the mask returned by the color allocation. Turning bits in this plane on (to 1's) changes the image to the fifth color, and turning them off reverts the image to its original color.

6. GRAPHICS AND TEXT

Graphics operations are often the most complex part of any window system, simply because so many different effects and variations are required to satisfy a wide range of applications. In this section we sketch the operations provided in X so that the basic level of graphics support can be understood. The operations are essentially a subset of the Digital Workstation Graphics Architecture; the VS100 display [14] implements this architecture for 1-bit pixel values. The set of operations was purposely kept simple in order to maximize portability.

Graphics operations in X are expressed in terms of relatively high-level concepts, such as lines, rectangles, curves, and fonts. This is in contrast to systems in which the basic primitives are to read and write individual pixels. Basing applications on pixel-level primitives works well when display memory can be mapped into the application's address space for direct manipulation. However, both display hardware and operating systems exist for which such

direct access is not possible, and emulating pixel-level manipulations in such an environment results in extremely poor performance. Expressing operations at a higher level avoids such device dependencies, as well as potential problems with network bandwidth. With high-level operations, a protocol request transmitted as a small number of bits over the network typically affects 10–100 times as many pixels on the screen.

6.1 Images

Two forms of offscreen images are supported in X: bitmaps and pixmaps. A bitmap is a single plane (bit) rectangle. A pixmap is an N -plane (pixel) rectangle, where N is the number of bits per pixel used by the particular display. A bitmap or pixmap can be created by transmitting all of the bits to the server; a pixmap can also be created by copying a rectangular region of a window. Bitmaps and pixmaps of arbitrary size can be created. Transmitting very large (or deep) images over a network connection can be quite slow; however, the ability to make use of shared memory in conjunction with the IPC mechanism would help enormously when the client and server are on the same machine.

The primary use of bitmaps is as masks (clipping regions). Several graphics requests allow a bitmap to be used as a clipping region, as in [37]. Bitmaps are also used to construct cursors, as described in Section 8. Pixmaps are used for storing frequently drawn images and as temporary backing-store for pop-up menus (as described in Section 8). However, the principal use of pixmaps is as tiles, that is, as patterns that are replicated in two dimensions to cover a region. Since there are often hardware restrictions as to what tile shapes can be replicated efficiently, guaranteed shapes are not defined by the X protocol. An application can query the server to determine what shapes are supported, although to date most applications simply assume 16-by-16 tiles are supported. A better semantics is to support arbitrary shapes but allow applications to query which shapes are most efficient.

The tiling origin used in X is almost always the origin of the destination window. That is, if enough tiles have been laid out, one tile would have its upper left corner at the upper left corner of the window. In this way, the contents of the window are independent of the window's position on the screen, and the window can be moved transparently to the application.

Servers vary widely in the amount of offscreen memory provided. For example, some servers limit offscreen memory to that accessible directly to the graphics processor (typically one to three times the size of screen memory), and fonts and other resources are allocated from this same pool. Other servers utilize their entire virtual address space for offscreen memory. Since offscreen memory for images is finite, an explicit part of the X protocol is the possibility that bitmap or pixmap creation can fail. Depending on the intended use of the image, the application may or may not be able to cope with the failure. For example, if the image is being stored simply to speed up redisplay, the application can always transmit the image directly each time (see below). If the image is to be a temporary backing-store for a window, the application can fall back on normal exposure processing (as described in Section 7). Servers should be constructed in such a way as to virtually guarantee sufficient memory (e.g., by caching images) for

creating at least small tiles and cursors, although this is not true in current implementations.

6.2 Graphics

All graphics and text requests include a logic function and a plane-select mask (an integer with the same number of bits as a pixel value) to modify the operation. All 16 logic functions are provided, although in practice only a few are ever used. Given a source and destination pixel, the function is computed bitwise on corresponding bits of the pixels, but only on bits specified in the plane-select mask. Thus the result pixel is computed as

((source FUNC destination), AND mask) OR (destination AND (NOT mask)).

The most common operation is simply replacing the destination with the source in all planes.

The simplest graphics request takes a single source pixel value and combines it with every pixel in a rectangular region of a window. Typically, this is used to fill a region with a color, but by varying the logic function or masks, other effects can be achieved. A second request takes a tile, effectively constructs a tiled rectangular source with it, and then combines the source with a rectangular region of a window.

An arbitrary image can be displayed directly, without first being stored off-screen. For monochrome images, the full contents of a bitmap are transmitted, along with a pair of pixel values; the image is displayed in a region of a window with those two colors. For color images, the full contents of a pixmap can be transmitted and displayed. In order to avoid requiring inordinate buffer space in the server, very large images must be broken into sections on the client side and displayed in separate requests.

The CopyArea request allows one region of a window to be moved to (or combined with) another region of the same window. This is the usual *bitblt*, or “bit block transfer” operation. The source and destination are given as rectangular regions of the window; the two regions have the same dimensions. The operation is such that overlap of the source and destination does not affect the result.

X provides a complex primitive for line drawing. It provides for arbitrary combinations of straight and curved segments, defining both open and closed shapes. Lines can be *solid*, by drawing with a single source pixel value, *dashed*, by alternately drawing with a single source pixel value and not drawing, and *patterned*, by alternately drawing with two source pixel values. Lines are drawn with a rectangular brush. Clients can query the server to determine what brush shapes are supported; a better semantics would be to support arbitrary shapes but allow applications to query which shapes are most efficient.

A final request allows an arbitrary closed shape (such as could be specified in the line-drawing request) to be filled with either a single source pixel value or a tile. For self-intersecting shapes, the even-odd rule is used: A point is inside the shape if an infinite ray with the point as origin crosses the path an odd number of times.

6.3 Text

For high-performance text, X provides direct support for bitmap fonts. A font consists of up to 256 bitmaps; each bitmap in a font has the same height but can vary in width. To allow server-specific font representations, clients “create” fonts by specifying a name rather than by downloading bitmap images into the server. An application can use an arbitrary number of fonts, but (as with all resources) font allocation can fail for lack of memory. A reasonably implemented server should support an essentially unbounded number of fonts (e.g., by caching), but some existing server implementations are deficient in this respect. Unlike Andrew [10], no heuristics are applied by the server when resolving a name to a font; specific communities or applications may demand a variety of heuristics, and as such they belong outside the base window system. Also unlike Andrew, the X server is not free to dynamically substitute one font for another; we do not believe such behavior is necessary or appropriate in the base window system.

A string of text can be displayed by using a font either as a mask or as a source. When a font is used as a mask, the foreground (the 1 bits in the bitmap) of each character is drawn with a single source pixel value. When a font is used as a source, the entire image of each character is drawn, using a pair of pixel values. Source font output is provided specifically for applications using fixed-width fonts in emulating traditional terminals.

To support “cut-and-paste” operations between applications, the server provides a number of buffers into which a client can read and write an arbitrary string of bytes. (This mechanism was adopted from Andrew.) Although these buffers are used principally for text strings, the server imposes no interpretation on the data, so cooperating applications can use the buffers to exchange such things as resource identifiers and images.

7. EXPOSURES

Given that output to obscured windows is possible, the issue of *exposure* must be addressed. When all (or a piece) of an obscured window again becomes visible (e.g., as the result of the window being raised), is the client or the server responsible for restoring the contents of the window? In X, it is the responsibility of the client. When a region of a window becomes exposed, the server sends an asynchronous event to the client specifying the window and the region that have been exposed; the rest is up to the application. A trivial application might simply redraw the entire window; a more sophisticated application would only redraw the exposed region.

Why is the client responsible? Because X imposes no structure on or relationships between graphics operations from a client, there are only two basic mechanisms by which the server might restore window contents: by maintaining display lists and by maintaining offscreen images. In the first approach, the server essentially retains a list of all output requests performed on the window. When a region of the window becomes exposed, the server reexecutes either all requests to the entire window or only requests that affect the region while clipping the output to that region. In the alternative approach, when a window

becomes obscured, the server saves the obscured region (or perhaps the entire window) in offscreen memory. All subsequent output requests are executed not only to the visible regions of the window, but to the offscreen image as well. When an obscured region becomes visible again, the offscreen copy is simply restored.

We believe that neither server-based approach is acceptable. With display lists, the server is unlikely to have any reasonable notion of when later output requests nullify earlier ones. Either the display list becomes unmanageably long, and a refresh that should appear nearly instantaneous instead appears as an extended replay, or the server spends a significant length of time pruning the display list, and normal-case performance is considerably reduced. One problem with the offscreen image approach is (virtual) memory consumption: On a 1024-by-1024 eight-plane display, just one full-screen image requires 1 megabyte (Mbyte) of storage, and multiple overlapping windows could easily require many times that amount. Another problem is that the cost of the implementation can be prohibitive. Consider, for example, the QDSS display [7], which has a graphics coprocessor. In the QDSS, display memory is inaccessible to the host processor. In addition, the coprocessor cannot perform operations in host memory and has relatively little offscreen memory of its own. The only viable way to maintain offscreen images for displays like the QDSS may be to emulate the coprocessor in software. It can easily take tens of thousands of lines of code to emulate a coprocessor, and such emulation may execute orders of magnitude slower than the coprocessor.

Our belief is that many applications can take advantage of their own information structures to facilitate rapid redisplay, without the expense of maintaining a distinct display structure or backing-store in the client or the server, and often with even better performance. (Sapphire [21] permits client refresh for this reason.) For example, a text editor can redisplay directly from the source, and a VLSI editor can redisplay directly from the layout and component definitions. Many applications will be built on top of high-level graphics libraries that automatically maintain the data structures necessary to implement rapid redisplay. For example, the structured display file mechanism in VGTS could be supported in a client library. Of course, pushing the responsibility back on the application may not simplify matters, particularly when retrofitting old systems to a new environment. For example, the current GKS design does not quite provide adequate hooks for automatic, system-generated refresh of application windows, nor does it provide an adequate mechanism for forcing refresh back on the application.

Relying on client-controlled refresh also derives from window management philosophy. Our belief is that applications cannot be written with fixed top-level window sizes built in. Rather, they must function correctly with almost any size and continue to function correctly as windows are dynamically resized. This is necessary if applications are to be usable on a variety of displays under a variety of window management policies. (Of course, an application may need a minimum size to function reasonably and may prefer the width or height to be a multiple of some number; X allows the client to attach a resize hint to each window to inform window managers of this.) Our belief is that most applications, for one

reason or another, will already have code for performing a complete redisplay of the window, and that it is usually straightforward to modify this code to deal with partial exposures. Similar arguments were used in the design of both Andrew and Mex and confirmed by experience [10, 25, 26].

This is not to argue that the server should never maintain window contents, only that it should not be *required* to maintain contents. For complex imaging and graphics applications, efficient maintenance by the server may be critical for acceptable performance of window management functions. There is nothing inherent in the X protocol that precludes the server from maintaining window contents and not generating exposure events. In the next version of X, windows will have several attributes to advise the server as to when and how contents should be maintained.

In X, clients are never informed of what regions are obscured, only of what regions have become visible. Thus, clients have insufficient information for optimizing output by only drawing to visible regions. However, we feel this is justified on two grounds. First, realistically, users seldom stack windows such that the active ones are obscured, so there is little point in complicating applications to optimize this case. More important, allowing applications to restrict output to only visible regions would conflict with the desire to have the server maintain obscured regions automatically when possible.

An interesting complication with the CopyArea request (described in Section 6) arises when client refresh is decided on. If part of the source region of the CopyArea is obscured, then not all of the destination region can be updated properly, and the client must be notified (with an exposure event) so that it can correct the problem. Since output requests are asynchronous, care must be taken by the application to handle exposure events when using CopyArea. In particular, if a region is exposed and an event sent by the server, a subsequent CopyArea may move all or part of the region before the event is actually received by the application. Several simple algorithms have been designed to deal with this situation, but we do not present them here.

Client refresh raises a visual problem in a network environment. When a region of a window becomes exposed, what contents should the server initially place in the window? In a local, tightly coupled environment, it might be perfectly reasonable to leave the contents unaltered, because the client can almost instantaneously begin to refresh the region. In a network environment, however (and even in a local system where processes can get “swapped out” and take considerable time to swap back in), inevitable delays can lead to visually confusing results. For example, the user may move a window and see two images of the window on the screen for a significant length of time, or resize a window and see no immediate change in the appearance of the screen.

To avoid such anomalies in X, clients must define a *background* for every window. The background can be a single color, or it can be a tiling pattern. Whenever a region of a window is exposed, the server immediately paints the region with the background. Users therefore see window shapes immediately, even if the “contents” are slow to arrive. Of course, many application windows have some notion of a background anyway, so having the server initialize with a background seldom results in extraneous redisplay. In fact, many nonleaf

windows typically contain nothing but a background, and having the server paint that background frees the applications from performing any redisplay at all to those windows.

Although we believe client-generated refresh is acceptable most of the time, it does not always perform well with momentary pop-up menus, where speed is at a premium. To avoid potentially expensive refresh when a menu is removed from the screen, a client can explicitly copy the region to be covered by the menu into offscreen memory (within the server) before mapping the menu window. A special unmap request is used to remove the menu: It unmaps the window without affecting the contents of the screen or generating exposure events. The original contents are then copied back onto the screen. In addition, the client usually *grabs* the server for the entire sequence, using a request that freezes all other clients until a corresponding ungrab request is issued (or the grabbing client terminates). Without this, concurrent output from other clients to regions obscured by the menu would be lost. Although freezing other clients is, in general, a poor idea, it seems acceptable for momentary menus.

8. INPUT

We now turn to a discussion of input events, but first we briefly describe the support for mouse cursors. Clients can define arbitrary shapes for use as mouse cursors. A cursor is defined by a source bitmap, a pair of pixel values with which to display the bitmap, a mask bitmap that defines the precise shape of the image, and a coordinate within the source bitmap that defines the “center” or “hot spot” of the cursor. Cursors of arbitrary size can be constructed, although only a portion of the cursor may be displayed on some hardware. Clients can query the server to determine what cursor sizes are supported, but existing applications typically just assume a 16-by-16 image can always be displayed. Cursors also can be constructed from character images in fonts; this provides a simple form of named indirection, allowing custom-tailoring to each display without having to modify the applications.

A window is said to *contain* the mouse if the hot spot of the cursor is within a visible portion of the window or one of its subwindows. The mouse is said to be *in* a window if the window, but no subwindow, contains the mouse. Every window can have a mouse cursor defined for it. The server automatically displays the cursor of whatever window the mouse is currently in; if the window has no cursor defined, the server displays the cursor of the closest ancestor with a cursor defined.

Input is associated with windows. Input to a given window is controlled by a single client, which need not be the client that created the window. Events are classified into various types, and the controlling client selects which types are of interest to it. Only events matching in type with this selection are sent to the client. When an input event is generated for a window and the controlling client has not selected that type, the server *propagates* the event to the closest ancestor window for which some client has selected the type, and sends the event to that client instead. Every event includes the window that had the event type selected; this window is called the *event window*. If the event has been propagated, the

event also includes the next window down in the hierarchy between the event window and the original window on which the event was generated.

8.1 The Keyboard

For the keyboard, a client can selectively receive events on the press or release of a key. Keyboard events are not reported in terms of ASCII character codes; instead, each key is assigned a unique code, and client software must translate these codes into the appropriate characters. The mapping from keycaps to keycodes is intended to be “universal” and predefined; a given keycap has the same keycode on all keyboards. Applications generally have been written to read a “keymap file” from the user’s home directory so that users can remap the keyboard as they see fit.

The use of coded keys is secondary to the ability to detect both up and down transitions on the keyboard. For example, a common trick in window systems is for mouse button operations to be affected by keyboard *modifiers* such as the Shift, Control, and Meta keys. A useful feature of the Genera [34] system is the use of a “mouse documentation line,” which changes dynamically as modifiers are pressed and released, indicating the function of the mouse buttons. A base window system must provide this capability. Transitions are not only useful on modifiers; various applications for systems other than X have been designed to use “chords” (groups of keys pressed simultaneously), and again the window system should support them.

The keyboard is always *attached* to some window (typically the root window or a top-level window); we call this window the *focus* window. A request can be used (usually by the input manager) to attach the keyboard to any window. The window that receives keyboard input depends on both the mouse position and the focus window. If the mouse is in some descendant of the focus window, that descendant receives the input. If the mouse is not in a descendant of the focus window, then the focus window receives the input, even if the mouse is outside the focus window. For applications that wish to have the mouse state modify the effect of keyboard input, a keyboard event contains the mouse coordinates, both relative to the event window and global to the screen, as well as the state of the mouse buttons.

To provide a reasonable user interface, keyboard events also contain the state of the most common modifier keys: Shift, ShiftLock, Control, and Meta. Without this information, anomalous behavior can result. If the user switches windows while modifier keys are down, the new client must somehow determine which modifiers are down. Placing the modifier state in the keyboard events solves such problems and also has another benefit: Most clients do not have to maintain their own shadow of the modifier state and so often can completely ignore key release events. However, there is a conflict between this server-maintained state and client-maintained keyboard mappings. In particular, clients cannot use nonstandard keys as modifiers or chords without the possibility of anomalies, such as those described above. We believe the correct solution (not yet supported in X) is for the server to maintain a bit mask reflecting the full state of the keyboard and allow clients to read this mask. An application using chords or

nonstandard modifiers would request the server to send this mask automatically whenever the mouse has entered the application's window.

8.2 The Mouse

The X protocol is (somewhat arbitrarily) designed for mice with up to three buttons. An application can selectively receive events on the press or release of each button. Each event contains the current mouse coordinates (both local to the window and global to the screen), the current state of all buttons and modifier keys, and a timestamp that can be used, for example, to decide when a succession of clicks constitutes a double or triple click. An application can also choose to receive mouse motion events, either whenever the mouse is in the window or only when particular buttons have also been pressed. The application cannot control the granularity of the reporting, nor is any minimum granularity guaranteed. In fact, typical server implementations make an effort to compact motion events in order to minimize system overhead and wired memory in device drivers. Thus X may not serve adequately for fine-grained tracking, such as in fast moving freehand drawing applications.

Even with motion compaction, servers can generate considerable numbers of motion events. If an application attempts to respond in real time to every event, it can easily get far behind relative to the actual position of the mouse. Instead, many applications simply treat motion events as hints. When a motion event is received, the event is simply discarded, and the client then explicitly queries the server for the current mouse position. While waiting for the reply, more motion events may be received; these are also discarded. The client then reacts on the basis of the queried mouse position. The advantage of this scheme over continuously polling the mouse position is that no CPU time is consumed while the mouse is stationary.

Clients can also receive an event each time the mouse enters or leaves a window. This can be particularly useful in implementing menus. For example, each menu item can be placed in a separate subwindow of the overall menu window. When the mouse enters a subwindow, the item is highlighted in some fashion (e.g., by inverting the video sense), and when the mouse leaves the window, the item is restored to normal. Implementing a menu in this manner requires considerably less CPU overhead than continuously polling the mouse, and also less overhead than using motion events, since most motion events would be within windows and thus uninteresting.

Owing to the nature of overlapping windows and because continuous tracking by the server is not guaranteed, the mouse may appear to move instantaneously between any pair of windows on the screen. Certainly, the window the mouse was in should be notified of the mouse leaving, and the window the mouse is now in should be notified of the mouse entering. However, all of the "in between" windows in the hierarchy may also be interested in the transition. This is useful in simplifying the structure of some applications and is necessary in implementing certain kinds of window managers and input managers. Thus, when the mouse moves from window A to window B, with window W as their closest (least) common ancestor, all ancestors of A below W also receive leave events, and all ancestors of B below W receive enter events.

It might be argued that, except for mouse motion events, events are infrequent enough for the server to always send all events to the client and eliminate the complexity of selecting events. However, some applications are written with interrupt-driven input; events are received asynchronously and cause the current computation to be suspended so that the input can be processed. For example, a text editor might use interrupt-driven input, with the normal computation being redisplay of the window. The receipt of extraneous input events (e.g., key release events) can cause noticeable “hiccups” in such redisplay.

9. INPUT AND WINDOW MANAGEMENT

There are two basic modes of keyboard management: *real-estate* and *listener*. In real-estate mode, the keyboard “follows” the mouse; keyboard input is directed to whatever window the mouse is in. In listener mode, keyboard input is directed to a specific window, independent of the mouse position. A few systems provide only real-estate mode [2], some only listener mode [11, 18, 21, 25, 33, 34], and a few provide both [10, 30], although the mode may not be changeable during a session. Both modes are supported in X, and the mode can be changed dynamically. Real-estate mode is the default behavior, with the root window as the focus window, as described in the previous section. An input manager can also make some other (typically top-level) window the focus window, yielding listener mode. Note, however, that, in listener mode in X, the client controlling the focus window can still get real-estate behavior for subwindows, if desired; this capability has proved useful in several applications.

The primary function of a window manager is reconfiguration: restacking, resizing, and repositioning top-level windows. The configuration of nested windows is assumed to be application specific, and under control of the applications. There are two broad categories of window managers: *manual* and *automatic*. A manual window manager is “passive” and simply provides an interface to allow the user to manipulate the desktop; windows can be resized and reorganized at will. The initial size and position of a window typically (but not always) are under user or application control. Automatic window managers are “active” and operate for the most part without human interaction; size and position at window creation and reconfiguration at window destruction are chosen by the system. Automatic managers typically tile the screen with windows such that no two windows overlap, automatically adjusting the layout as windows are created and destroyed. Several systems [10, 18, 27, 36] provide automatic management plus limited manual reconfiguration capability.

Existing window managers for X are manual. Automatic management that is transparent to applications cannot be accomplished reasonably in X; future support for automatic management is discussed in Section 10. In the current X design, clients are responsible for initially sizing and placing their top-level windows, not window managers. In this way, applications continue to work when no window manager is present. Typically, the user either specifies geometry information in the application command line or uses the mouse to sweep out a rectangle on the screen. (For the latter, the application grabs the mouse, as described below.)

9.1 Mouse-Driven Management

Existing managers are primarily mouse driven and are based on the ability to “steal” events. Specifically, a manager (or any other client) can *grab* a mouse button in combination with a set of modifier keys, with the following effect: Whenever the modifier keys are down and the button is pressed, the event is reported to the grabbing client, regardless of what window the mouse is in. All mouse-related events continue to be sent to that client until the button is released. As part of the grab, the client also specifies a mouse cursor to be used for the duration of the grab and a window to be used as the event window. A manager specifies the root window as the event window when grabbing buttons; with the event propagation semantics described in Section 8, the grabbed events contain not only the global mouse coordinates, but also the top-level application window (if any) containing the mouse. This is sufficient information to manipulate top-level windows.

This button-grab mechanism has enabled several different management interfaces to be built, including a “programmable” interface [8] that allows the user to assign individual commands or user-defined menus of commands to any number of button/modifier combinations. For example, a button click (press and release without intervening motion) might be interpreted as a command to raise or lower a window, or to attach the keyboard; a press/motion/release sequence might be interpreted as a command to move a window to a new position; or a button press might cause a menu to pop up, with the selection indicated by the mouse position at the release of the button. By allowing both specific commands and menus to be bound to buttons, a range of interfaces can be constructed to satisfy both “expert” and “novice” users.

Another form of manager simply displays a static menu bar along the top of the screen, with items for such operations as moving a window and attaching the keyboard. The menu is used in combination with a mouse-grab primitive, with which a client can unilaterally grab the mouse and then later explicitly release it; during such a mouse grab, events are redirected to the grabbing client, just as for button grabs. When the user clicks on a menu bar item with any button, the manager unilaterally grabs the mouse. The user then uses the mouse to execute the specific command. For example, having clicked on the “move” item, the user indicates the window to be moved by placing the mouse in the window and pressing a button and then indicates the new position by moving the mouse and releasing the button. The manager then releases the mouse.

9.2 Icons

One important “resizing” operation performed by a window manager is transforming a window into a small icon and back again. In X, icons are merely windows. Transforming a window into an icon simply involves unmapping the window and mapping its associated icon. The association between a window and its icon is maintained in the server, rather than in the window manager, and either the application or the manager can provide the icon. In this way, the manager can provide a default icon form for most clients, but clients can provide their own if desired, possibly with dynamic rather than static contents. The client is still insulated from management policy, even if it provides the icon: The

manager is responsible for positioning, mapping, and unmapping the icon, and the client is responsible only for displaying the contents.

The icon state is maintained in the server not only to allow clients to provide icons, but to avoid the loss of state if the window manager should terminate abnormally. When a window manager terminates, any windows it has created are destroyed, including icon windows. With knowledge of icons, the server can detect when an icon is destroyed and automatically remap the associated client window. Without this, abnormal termination of the window manager would result in “lost” windows.

9.3 Race Conditions

There are many race conditions that must be dealt with in input and window management because of the asynchronous nature of event handling. For example, if a manager attempts to grab the mouse in response to a press of a button, the mouse-grab request might not reach the server until after the button is released, and intervening mouse events would be missed. Or, if the user clicks on a window to attach the keyboard there and then immediately begins typing, the first few keystrokes might occur before the manager actually responds to the click and the server actually moves the keyboard focus. A final example is a simple interface in which clicking on a window lowers it. Given a stack of three windows, the user might rapidly click twice in the same spot, expecting the top two windows to be lowered. Unless the first click is sent to the manager and the resulting request to lower is processed by the sever before the second click takes place, the event window for the second click will be the same as for the first click, and the manager will lower the first window twice.

A work-around for the last example, used by existing managers, is to ignore the event window reported in most events. Instead, the global mouse coordinates reported in the event are used in a follow-up query request to determine which top-level window now contains that coordinate. However, not all race conditions have acceptable solutions within the current X design. For a general solution it must be possible for the manager to synchronize operations explicitly with event processing in the server. For example, a manager might specify that, at the press of a button, event processing in the server should cease until an explicit acknowledgment is received from the manager.

10. FUTURE

On the basis of critiques from numerous universities and commercial firms, fairly extensive evaluation and redesign of the X protocol have been under way since May 1986. Our desire is to define a “core” protocol that can serve as a standard for window system construction over the next several years. We expect to present the rationale for this new design in the very near future, once it has been validated by at least a preliminary implementation. In this section, we highlight the major protocol changes.

10.1 Resource Allocation

Since the server is responsible for assigning identifiers to resources, each resource allocation currently requires a round-trip time in order to perform. For applications that allocate many resources, this causes a considerable start-up

delay. For example, a multipane menu might consist of dozens of windows, numerous fonts, and several different mouse cursors, leading to a delay of 1 second or longer.

In retrospect, this is the most significant defect in the design of X. To get around these delays, programming interfaces have been augmented to provide “batch mode” operations. If several resources must be created, but there are no interdependencies among the allocation requests, all of the requests are sent in a batch, and then all of the replies are received. This effectively reduces the delay to a single round-trip time.

A better solution to this problem is to make clients generate the identifiers. When the client establishes a connection to the server, it is given a specific subrange from which it can allocate. This change will significantly improve start-up times without affecting applications, as identifiers can be generated inside low-level libraries without changing programming interfaces.

10.2 Transparent Windows

Transparent windows can be used as clipping regions; however, they are unsatisfactory for this purpose because every coordinate in a graphics request must be translated by the client from the “real” window’s origin to the transparent window’s origin. A better approach to clipping regions is to allow clients to create clipping regions and attach them to all graphics requests. As noted in Section 6, X currently allows a clipping region in the form of a bitmap to be attached to a few graphics requests. Allowing a clipping region, specified either as a bitmap or a list of rectangles, to be attached to all graphics requests provides a more uniform mechanism.

To date transparent windows have been primarily used as inexpensive opaque windows. In the current server implementation, transparent windows can be created and transformed significantly faster than opaque windows. Because of this, transparent windows are often used when opaque windows would otherwise be adequate. We believe a new implementation of the server will improve the performance of opaque windows to the point at which this will no longer be necessary.

With explicit clipping regions added for graphics and the performance advantages of transparent windows reduced, the only remaining use of transparent windows is for input (and cursor) control. Various applications want relatively fine-grained input control, and such control must not affect graphics output. Close control of cursor images and mouse motion events seems particularly important. However, the vast majority of the time control naturally is associated with normal window boundaries, so it would be unwise to divorce input control completely from windows. As such, the new protocol provides “input-only” windows, which act like normal windows for the purposes of input and cursor control, but which cannot be used as a source or destination in graphics requests, and which are completely invisible as far as output is concerned.

10.3 Color

X originally was not designed to deal with direct-color displays. Direct-color displays typically have between 12 and 36 bits per pixel; the pixel value consists of three subfields, which are used as indexes into three independent color maps:

one for red intensities, one for green, and one for blue. Some direct-color displays also have a fourth subfield, sometimes referred to as “z-channel” information, used to control attributes such as blending or chroma keying. We now understand how to incorporate direct-color displays without z-channel information into X in such a way that the differences between direct-color and pseudocolor color maps need not be apparent to the application, yet still allow all of the usual color map tricks to be played.

At present there is only one color map for all applications, and color applications fail when this map gets full. Although dozens of applications typically can be run under X within a single 8-bit pseudocolor map, a single map is clearly unacceptable when dealing with small color maps or with multiple applications (e.g., CAD tools) that need large portions of the color map. The solution is to support multiple virtual color maps, still permitting applications to coexist within any map, but allowing the possibility that not all applications show true color simultaneously. This also matches next-generation displays, which actually support multiple color maps in hardware [39].

10.4 Graphics

Perhaps the biggest mistake in the graphics area was failing to support fonts with kerning (side bearings) [26]. For example, a relatively complete emulation of the Andrew programming interface was built for X, but Andrew applications depend heavily on kerned fonts. There are other deficiencies that will be corrected. For example, large glyph-sets (e.g., Japanese) will be supported, as well as stippling (using a clip mask constructed by tiling a region with a bitmap). The notions of line width, join style, and end style found in PostScript [1] are usually preferred to brush shapes for line drawing and will be supported.

In an attempt to support a wide range of devices, the exact path followed for lines and filled shapes was originally left undefined in X (the class of curve was not even specified). Different devices use slightly different algorithms to draw straight lines, and it seemed better to have high performance with minor variation than to have uniformity with poor performance. Relatively few devices support curve drawing in hardware, but some support it in firmware, and again performance seemed more important than accuracy. In retrospect, however, allowing such device-dependent behavior was a poor decision. The vast majority of applications draw lines aligned on an axis, and speed and precision are not an issue. The applications that do require complex shapes also require predictable results, so precise specifications are important.

A notable feature missing in X is the ability to perform graphics offscreen. The reasons for this are essentially the same as those presented in the discussion of exposures (Section 7). In particular, not all graphics coprocessors can operate on host memory, and emulating such processors can be expensive. However, application builders have demanded this capability, and the demand appears to be sufficient leverage for convincing server implementors to provide the capability. Offscreen graphics will be possible in the new protocol, although the amount of offscreen memory and its performance characteristics may vary widely. In addition, the protocol is being extended to allow the manipulation of both images and windows of varying depths. For example, a server might support depths of 1, 4, 8, 12, and 24 bits. This allows imaging applications to transmit data more

compactly, allows for more efficient memory utilization in the server, and provides a match with next-generation display hardware.

A common debate in graphics systems is whether and where to have state. Should parameters such as logic function, plane mask, source pixel value or tile, tiling origin, font, line width and style, and clipping region be explicit in every request or collected into a state object? The current X protocol is stateless for the following reasons: Both state and stateless programming interfaces can be easily built on top of the protocol; the currently supported graphics requests have just few enough parameters for them to be represented compactly; and the initial set of displays we were interested in (and the implementations we had in mind for them) would not benefit from the addition of state. However, we now believe that a state-based protocol is generally superior, since it handles complex graphics gracefully and allows significantly faster implementations on some displays.

10.5 Management

An obvious interface style presently not supported in X is the ability to use the keyboard for management commands. To allow this, a key-grab mechanism, akin to the button-grab mechanism described in Section 9, will be provided. To allow such styles as using the first button click in a window to attach the keyboard, both button grabs and key grabs have been extended to apply to specific subhierarchies, rather than always to the entire screen. To handle the kinds of race conditions described in Section 9, a general event synchronization mechanism has been incorporated into the grab mechanisms.

To support automatic window management, a manager must be able to intercept certain management requests from clients (such as mapping or moving a window) before they are executed by the server, and to be notified about others (such as unmapping a window) after they are executed. In addition, some managers want to provide uniform title bars and border decorations automatically. To allow this, it is useful to be able to “splice” hierarchies: to move a window from one parent to another. To allow input managers and window managers to be implemented as separate applications, the ability for multiple clients to select events on the same window is being added. For example, both a window manager and an input manager might be interested in the unmapping or destruction of a window.

10.6 Extensibility

The information that input and window managers might desire from applications is quite varied, and it would be a mistake to try to define a fixed set. Similarly, the information paths between applications (e.g., in support of “cut and paste”) need to be flexible. To this end, we are adding a LISPish property list [29] mechanism to windows, and the event mechanism is being augmented to provide a simple form of interclient communication.

The new X protocol explicitly continues to avoid certain areas, such as 3-D graphics and antialiasing. However, a general mechanism has been designed to allow extension libraries to be included in a server. The intention is that all servers implement the “core” protocol, but each server can provide arbitrary extensions. If an extension becomes widely accepted by the X community, it can

be adopted as part of the core. Each extension library is assigned a global name, and an application can query the server at run time to determine whether a particular extension is present. Request opcodes and event types are allocated dynamically, so that applications need not be modified to execute in each new environment.

11. SUMMARY

The X Window System provides high-performance, high-level, device-independent graphics. A hierarchy of resizable, overlapping windows allows a wide variety of application and user interfaces to be built easily. Network-transparent access to the display provides an important degree of functional separation, without significantly affecting performance, that is crucial to building applications for a distributed environment. To a reasonable extent, desktop management can be custom-tailored to individual environments, without modifying the base system and typically without affecting applications.

To date, the X design and implementation effort has focused on the base window system, as described in this paper, and on essential applications and programming interfaces. The design of the network protocol and the color allocation mechanism, the design and implementation of device-independent layer of server, and the implementation of several applications and a prototype window manager were carried out by the first author. The design and implementation of the C programming interface, the implementation of major portions of several applications, and the coordination of efforts within Project Athena and Digital Equipment Corporation were carried out by the second author. In addition, many other people from Project Athena, the Laboratory for Computer Science, and institutions outside MIT have contributed software.

Necessary applications, such as window managers and VT100 and Tektronics 4014 terminal emulators, have been created, and numerous existing applications, such as text editors and VLSI layout systems, have been ported to the X environment. Although several different menu packages have been implemented, we are only now beginning to see a rich library of tools (scroll bars, frames, panels, more menus, etc.) for facilitating the rapid construction of high-quality user interfaces. Tool building is taking place at many sites, and several universities are now attempting to unify window systems work with X as a base, so that such tools can be shared.

The use of X has grown far beyond anything we had imagined. Digital has incorporated X into a commercial product, and other manufacturers are following suit. With the appearance of such products and the release of complete X sources on the Berkeley 4.3 UNIX distribution tapes, it is no longer feasible to track all X use and development. Existing applications written in C are known to have been ported to 7 machine architectures of more than 12 manufacturers, and the C server to 6 machine architectures and more than 16 display architectures. In most cases the code is running under UNIX, but other operating systems are also involved. In addition, relatively complete server implementations exist in two LISP dialects. Apart from the portability of the system's design, a large part of this success is due to MIT's decision to distribute X sources without any

licensing restrictions, and the willingness of people in both educational and commercial institutions to contribute code without restrictions.

ACKNOWLEDGMENTS

Our thanks go to the many people who have contributed to the success of X. Particular thanks go to those who have made significant contributions to the nonproprietary implementation: Paul Asente (Stanford University), Scott Bates (Brown University), Mike Braca (Brown), Dave Bundy (Brown), Dave Carver (Digital), Tony Della Fera (Digital), Mike Gancarz (Digital), James Gosling (Sun Microsystems), Doug Mink (Smithsonian Astrophysical Observatory), Bob McNamara (Digital), Ron Newman (MIT), Ram Rao (Digital), Dave Rosenthal (Sun), Dan Stone (Brown), Stephen Sutphen (University of Alberta), and Mark Vandevoorde (MIT).

Special thanks go to Digital Equipment Corporation. A redesign of the protocol and a reimplementaion of the server to deal with color and to increase performance were made possible with funding (in the form of hardware) from Digital. To their credit, all of the resulting device-independent code remained the property of MIT.

REFERENCES

1. ADOBE SYSTEMS. *PostScript Language Reference Manual*. Addison-Wesley, Reading, Mass., 1985.
2. APOLLO COMPUTER. *Domain System User's Guide*. Apollo Computer, Chelmsford, Mass., 1985.
3. ASETE, P. W reference manual. Internal document, Dept. Computer Science, Stanford Univ., Calif., 1984.
4. BALKOVICH, E., LERMAN, S., AND PARMELEE, R. P. Computing in higher education: The Athena experience. *Commun. ACM* 28, 11 (Nov. 1985), 1214-1224.
5. CHERITON, D. The V kernel: A software base for distributed systems. *IEEE Softw.* 1, 2 (Apr. 1984), 19-42.
6. COHEN, D. On holy wars and a plea for peace. *Computer* 14, 10 (Oct. 1981), 48-54.
7. DIGITAL EQUIPMENT CORP. *VCB02 Video Subsystem Technical Manual*. Educational Services, Digital Equipment Corporation, Bedford, Mass., 1986.
8. GANCARZ, M. UWM: A user interface for X windows. In *Summer Conference Proceedings* (Atlanta, Ga., June 10-13). USENIX Association, 1986, pp. 429-440.
9. GETTYS, J. Problems implementing window systems in Unix. In *Winter Conference Proceedings* (Denver, Colo., Jan. 15-17). USENIX Association, 1986, pp. 89-97.
10. GOSLING, J., AND ROSENTHAL, D. A window-manager for bitmapped displays and Unix. In *Methodology of Window-Managers*, F. R. A. Hopgood et al., Eds. Springer-Verlag, New York, 1986.
11. HAWLEY, M. J., AND LEFFLER, S. J. Windows for Unix at Lucasfilm. In *Summer Conference Proceedings* (Portland, Oreg., June 11-14). USENIX Association, 1985, pp. 393-406.
12. INTERNATIONAL STANDARDS ORGANIZATION. Information processing: Graphical kernel system (GKS)—Functional description. Rep. DIS 7942, International Organization for Standardization, Geneva, Switzerland, 1982.
13. LANTZ, K. A., AND NOWICKI, W. I. Structured graphics for distributed systems. *ACM Trans. Graph.* 3, 1 (Jan. 1984), 23-51.
14. LEVY, H. VAXstation: A general-purpose raster graphics architecture. *ACM Trans. Graph.* 3, 1 (Jan. 1984), 70-83.
15. LIPKIE, D. E., EVANS, S. R., NEWLIN, J. K., AND WEISSMAN, R. L. Star graphics: An object-oriented implementation. *Comput. Graph.* 16, 3 (July 1982), 115-124.
16. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.* 5, 3 (July 1983), 381-404.

17. MCKEE, L. *MC-WINDOWS Programming Manual, Revision A*. Massachusetts Computer Corporation, Westford, Mass., 1985.
18. MICROSOFT CORP. *Microsoft Windows: Programmer's Guide*. Microsoft Corporation, Redmond, Wash., 1985.
19. MOON, D. Chaosnet. AI Memo 628, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., June 1981.
20. MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. H., AND DONELSON SMITH, F. Andrew: A distributed personal computing environment. *Commun. ACM* 29, 3 (Mar. 1986), 184–201.
21. MYERS, B. Issues in window management design and implementation. In *Methodology of Window-Managers*, F. R. A Hopgood et al., Eds. Springer-Verlag, New York, 1986.
22. NOWICKI, W. Partitioning of function in a distributed graphics system. Ph.D. dissertation, Dept. Computer Science, Stanford Univ., Calif., 1985.
23. PIKE, R. The Blit: A multiplexed graphics terminal. *AT&T Bell Lab. Tech. J.* 63, 8 (Oct. 1984), 1607–1631.
24. POSTEL, J. Transmission control protocol. Rep. RFC 793, USC/Information Sciences Institute, Marina del Rey, Calif., Sept. 1981.
25. RHODES, R., HAEBERLI, P, AND HICKMAN, K. Mex—A window manager for the IRIS. In *Summer Conference Proceedings* (Portland, Oreg., June 11–14). USENIX Association, 1985, pp. 381–392.
26. ROSENTHAL, D. Window system implementations. USENIX Association, 1986. (Course notes for *Winter Conference*, Denver.)
27. SMITH, D. C., IRBY, C., KIMBALL, R., AND HARSLEM, E. The Star user interface: An overview. In *Proceedings of the 1982 National Computer Conference* (Houston, Tex., June 7–10). AFIPS Press, Reston, Va., 1982, pp. 515–528.
28. STALLMAN, R., MOON, D., AND WEINREB, D. *Lisp Machine Window System Manual*. MIT Artificial Intelligence Laboratory, Cambridge, Mass., Aug. 1983.
29. STEELE, G. L. *Common Lisp: The Language*. Digital Press, Bedford, Mass., 1984.
30. SUN MICROSYSTEMS. *Programmer's Reference Manual for SunWindows*. Sun Microsystems, Mountain View, Calif., 1985.
31. SUN MICROSYSTEMS. *NeWS Preliminary Technical Overview*. Sun Microsystems, Mountain View, Calif., 1986.
32. SWEET, R. Mesa programming environment. *ACM SIGPLAN Not.* 20, 7 (July 1985), 216–229.
33. SWEETMAN, D. A modular window system for Unix. In *Methodology of Window-Managers*, F. R. A. Hopgood et al., Eds. Springer-Verlag, New York, 1986.
34. SYMBOLICS. *Programming the User Interface*. Symbolics, Cambridge, Mass., 1986.
35. TEITELMAN, W. The Cedar programming environment: A midterm report and examination. Rep. CSL 83-11, Xerox PARC, Palo Alto, Calif., June 1984.
36. TRAMMEL, R. D. A capability based hierarchic architecture for Unix window management. In *Summer Conference Proceedings* (Portland, Oreg., June 11–14). USENIX Association, 1985, pp. 373–379.
37. WARNOCK, J., AND WYATT, D. K. A device independent graphics imaging model for use with raster devices. *Comput. Graph.* 16, 3 (July 1982), 313–319.
38. WECKER, S. DNA: The digital network architecture. *IEEE Trans. Commun.* COM-28, 4 (Apr. 1980), 510–526.
39. WILKES, A. J., SINGER, D. W., GIBBONS, J. J., KING, T. R., ROBINSON, P., AND WISEMAN, N. E. The Rainbow workstation. *Comput. J.* 27, 2 (May 1984), 112–120.

Received July 1986; revised October 1986; accepted October 1986

EXHIBIT 2

United States Patent [19]

Agarwal

[11] Patent Number: 4,688,167

[45] Date of Patent: Aug. 18, 1987

- [54] SCREEN MANAGER FOR DATA PROCESSING SYSTEM
- [75] Inventor: Arun K. Agarwal, Chelmsford, Mass.
- [73] Assignee: Wang Laboratories, Inc., Lowell, Mass.
- [21] Appl. No.: 655,280
- [22] Filed: Sep. 27, 1984
- [51] Int. Cl.⁴ G06F 3/14
- [52] U.S. Cl. 364/200; 340/734; 340/747
- [58] Field of Search ... 364/200 MS File, 900 MS File, 364/300; 340/709, 712, 734, 735, 747

[56] References Cited

U.S. PATENT DOCUMENTS

| | | | |
|-----------|---------|-----------------|-----------|
| 4,189,727 | 2/1980 | Vaughn, Jr. | 340/711 |
| 4,475,156 | 10/1984 | Federico et al. | 364/300 |
| 4,484,302 | 11/1984 | Cason et al. | 364/900 |
| 4,533,910 | 8/1985 | Sukonick et al. | 340/721 |
| 4,550,315 | 10/1985 | Bass et al. | 340/703 |
| 4,555,775 | 11/1985 | Pike | 340/734 X |
| 4,559,533 | 12/1985 | Bass et al. | 340/747 X |
| 4,586,158 | 4/1986 | Brandle et al. | 364/900 |

FOREIGN PATENT DOCUMENTS

0121015 10/1984 European Pat. Off. .

OTHER PUBLICATIONS

Robins et al., "Viewporting in an Alphanumeric Dis-

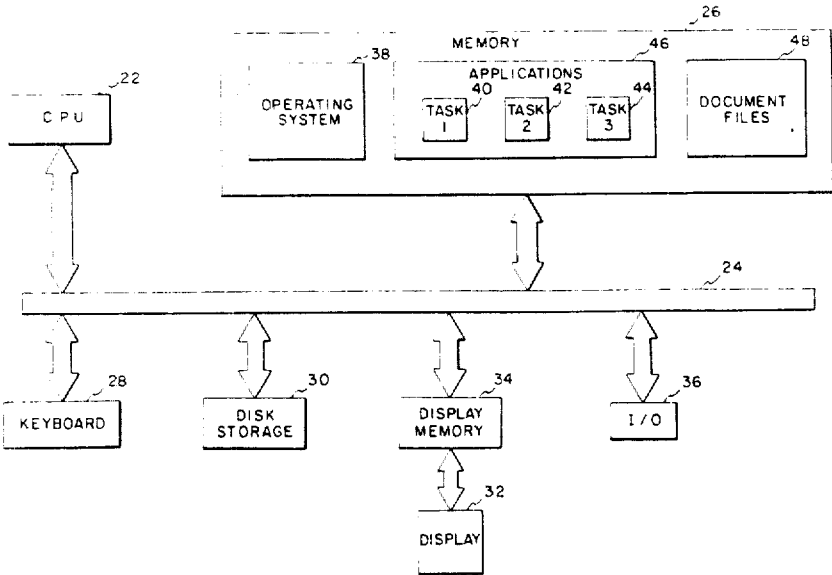
play", IBM Tech. Disc. Bull. vol. 20, vol. 10, 3/78, pp. 4148-4151.

Primary Examiner—Raulfe B. Zache
Attorney, Agent, or Firm—Michael H. Shanahan; Gary D. Clapp

[57] ABSTRACT

In a multi-tasking data processing system, each task may request that the operating system set up descriptor blocks which identify virtual screens for display of data on the video display. Under keyboard control, only one virtual screen is selected for display at a given time. The operating system reserves a portion of the video display for displaying identifiers of the virtual screens which have been established but which are held in background. Each virtual screen may be subdivided into viewports by the corresponding application task. Those viewports are also identified in the operating system by descriptor blocks which point to pages of data in the document files. The descriptor blocks can be modified through requests from application tasks even when held in background. Whenever the display memory is updated, data designated by the descriptor blocks is passed through a rasterizer in the operating system which generates the pixel data to be stored in a display memory.

43 Claims, 11 Drawing Figures



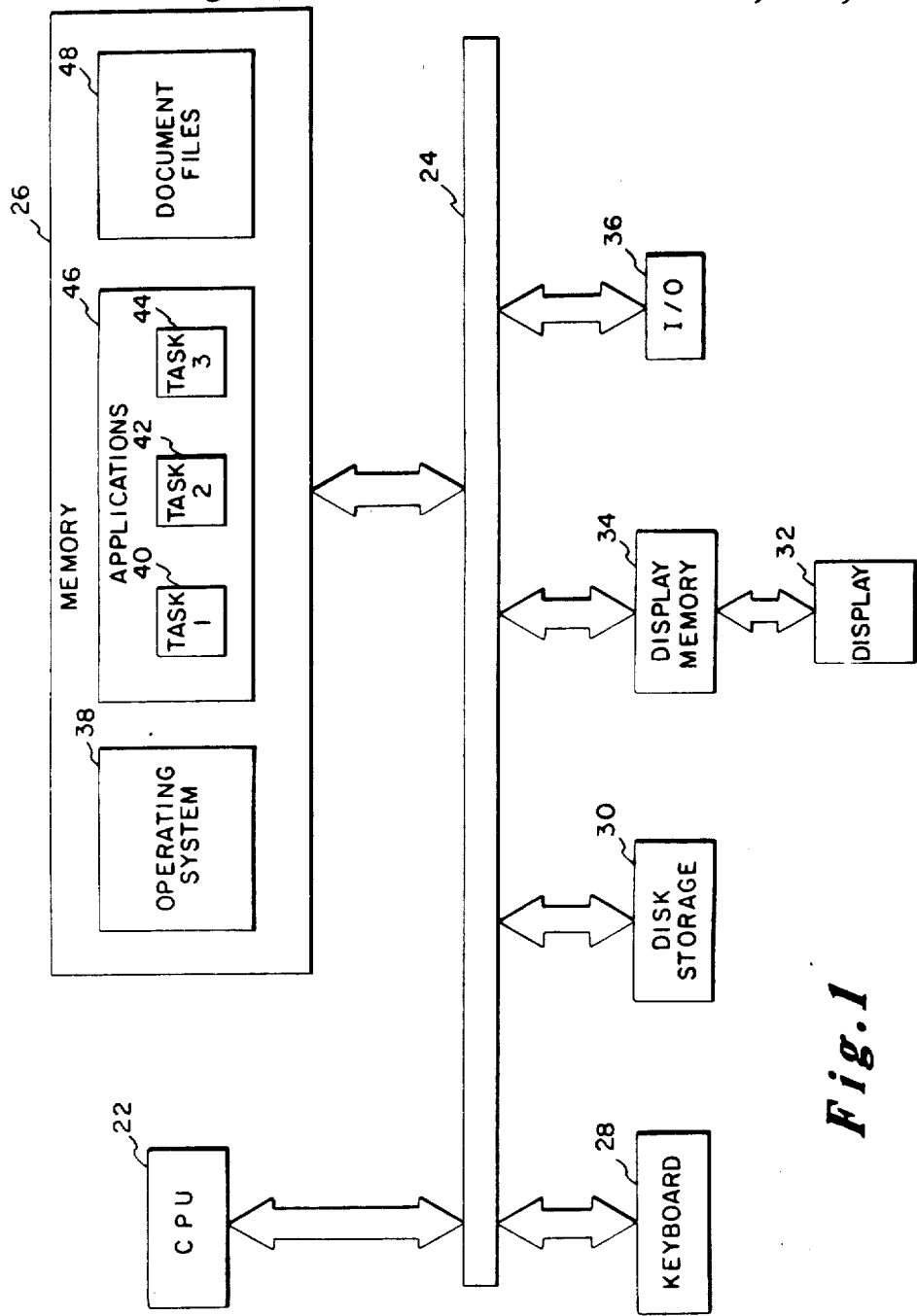


Fig. 1

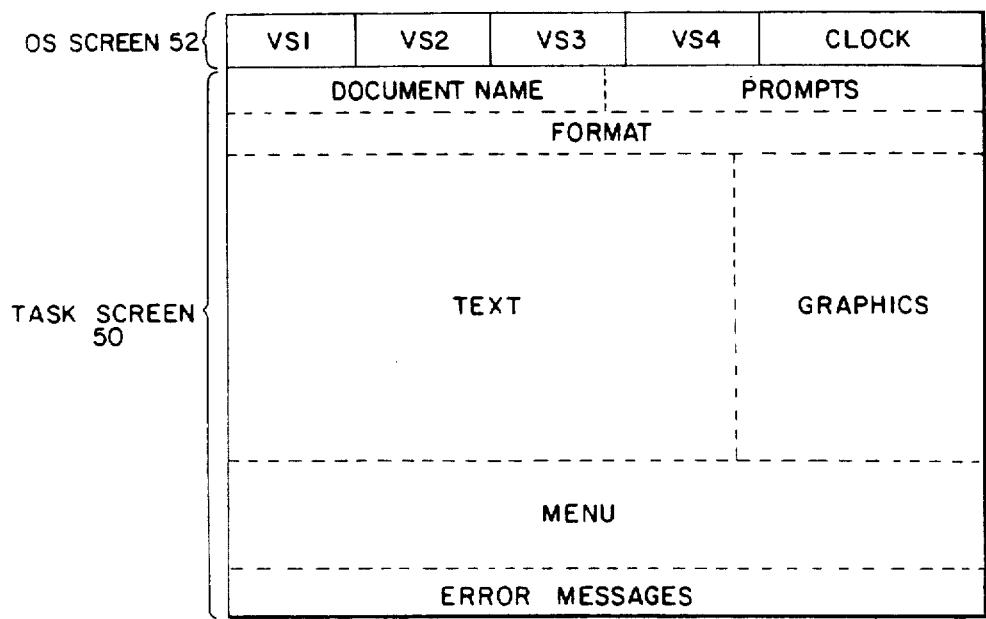


Fig. 2

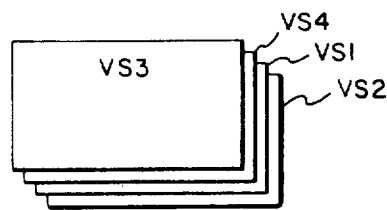


Fig. 3

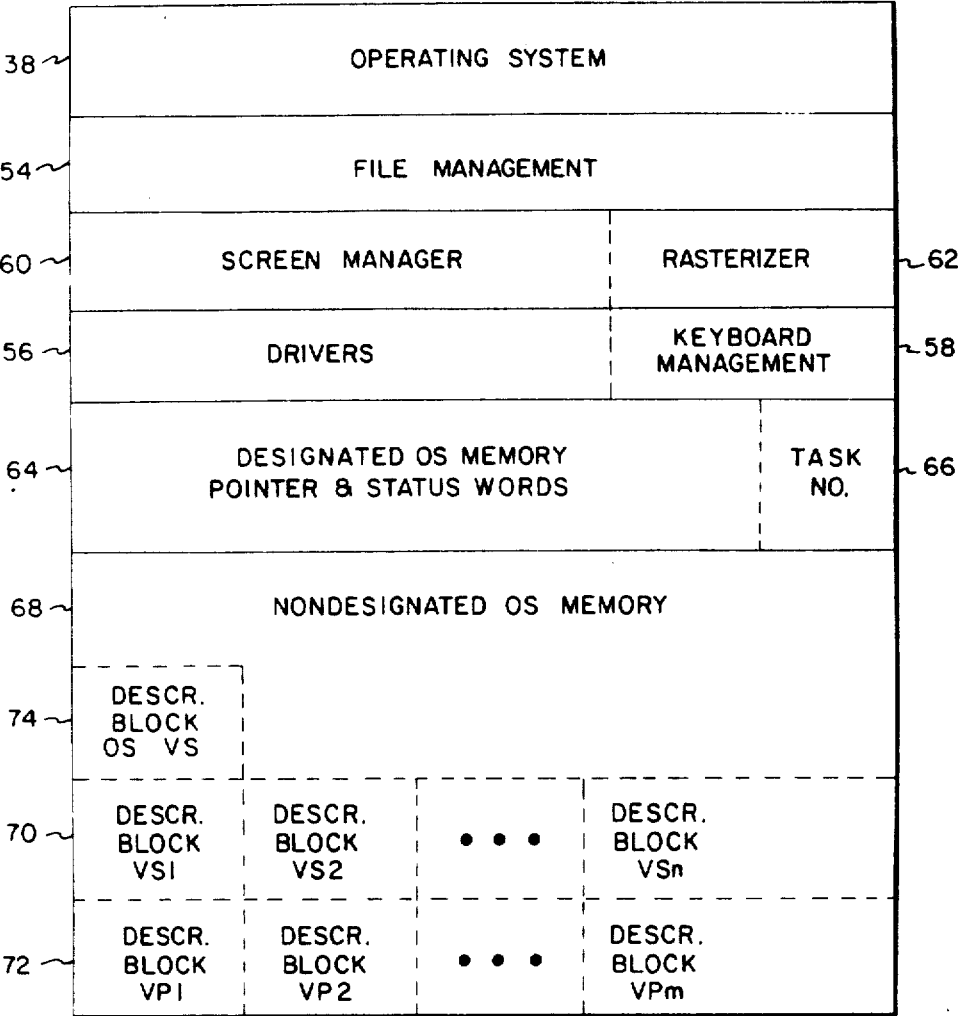


Fig.4

DATA STRUCTURES
SHEET 1

| | |
|-------------------|-------------------|
| FIG. 5 SHEET 1 | FIG. 5 SHEET 2 |
|-------------------|-------------------|

KEY

OPERATING SYSTEM

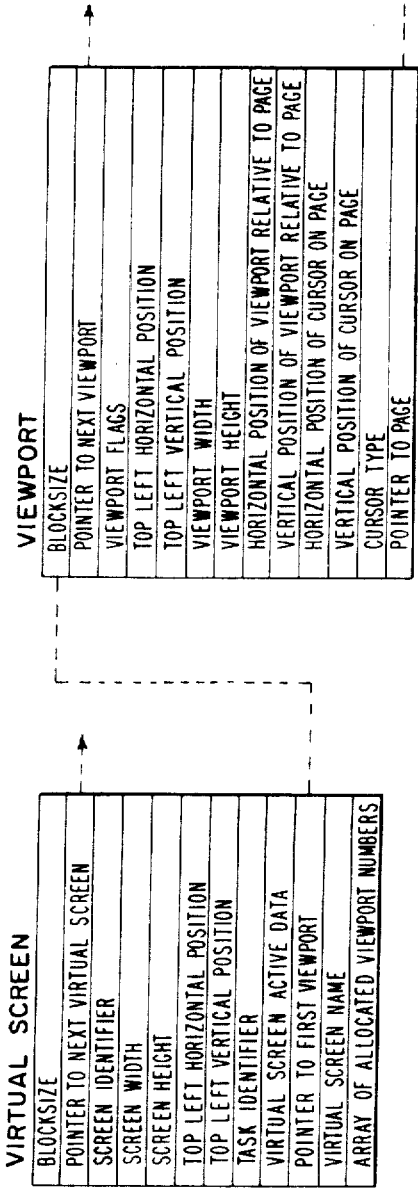


Fig. 5
SHEET 1

DATA STRUCTURES
SHEET 2

DOCUMENT FILES

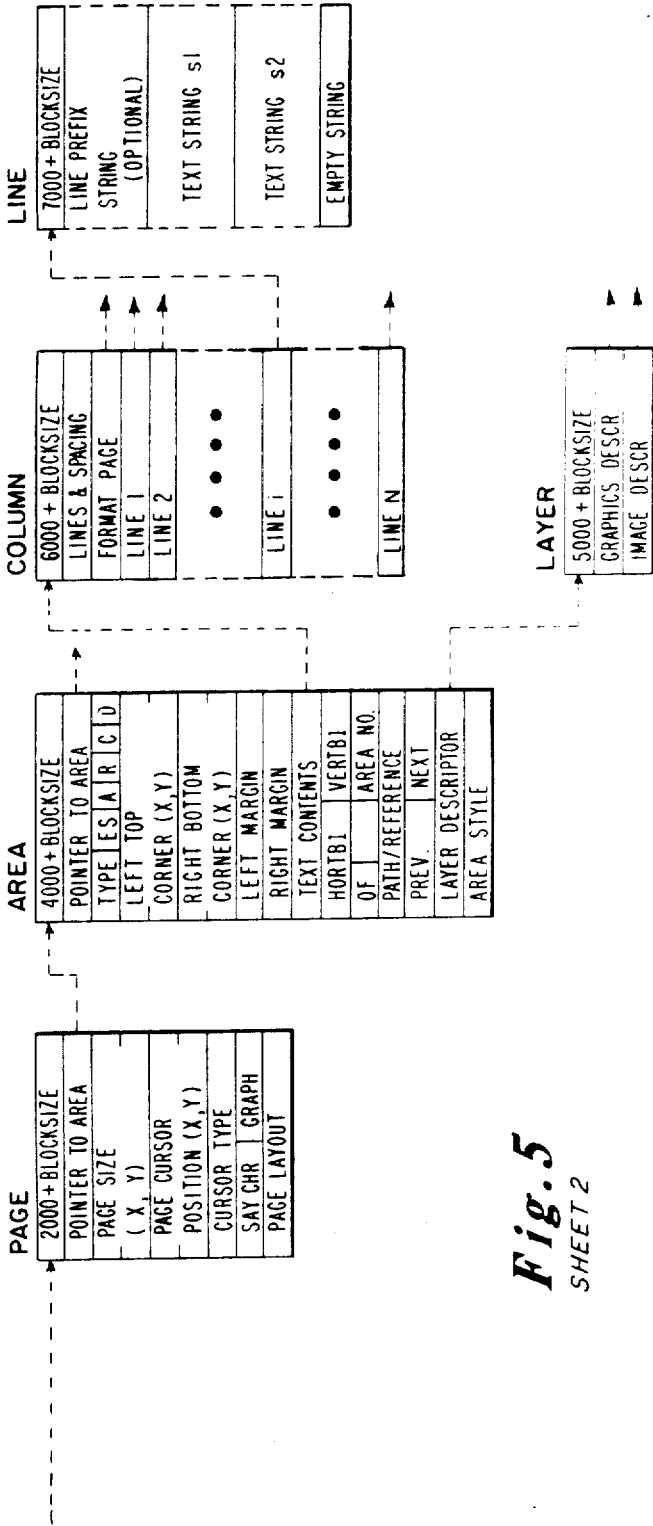


Fig. 5
SHEET 2

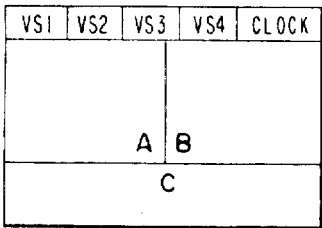


Fig. 6A

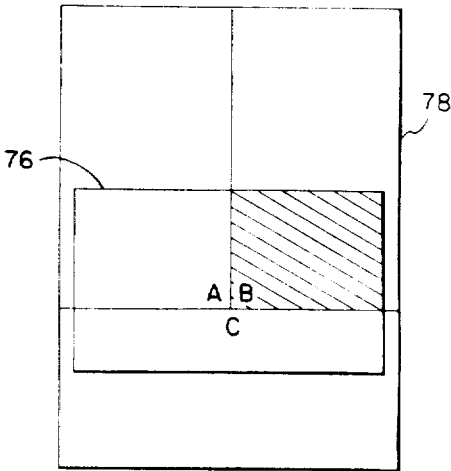
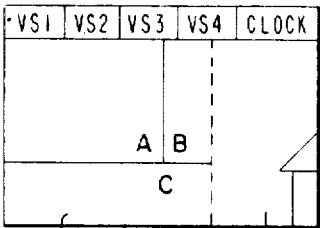


Fig. 6B



76 80

Fig. 7A

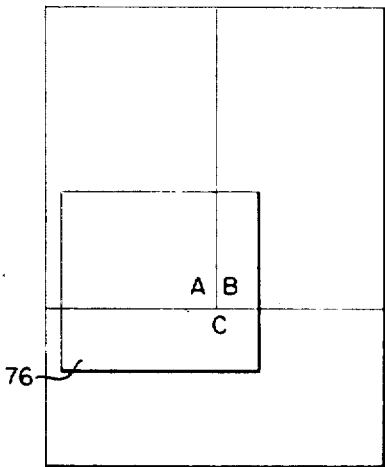


Fig. 7B

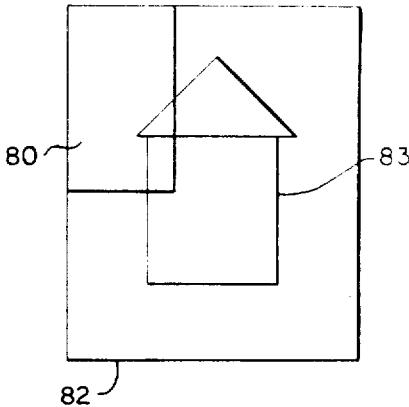


Fig. 7C

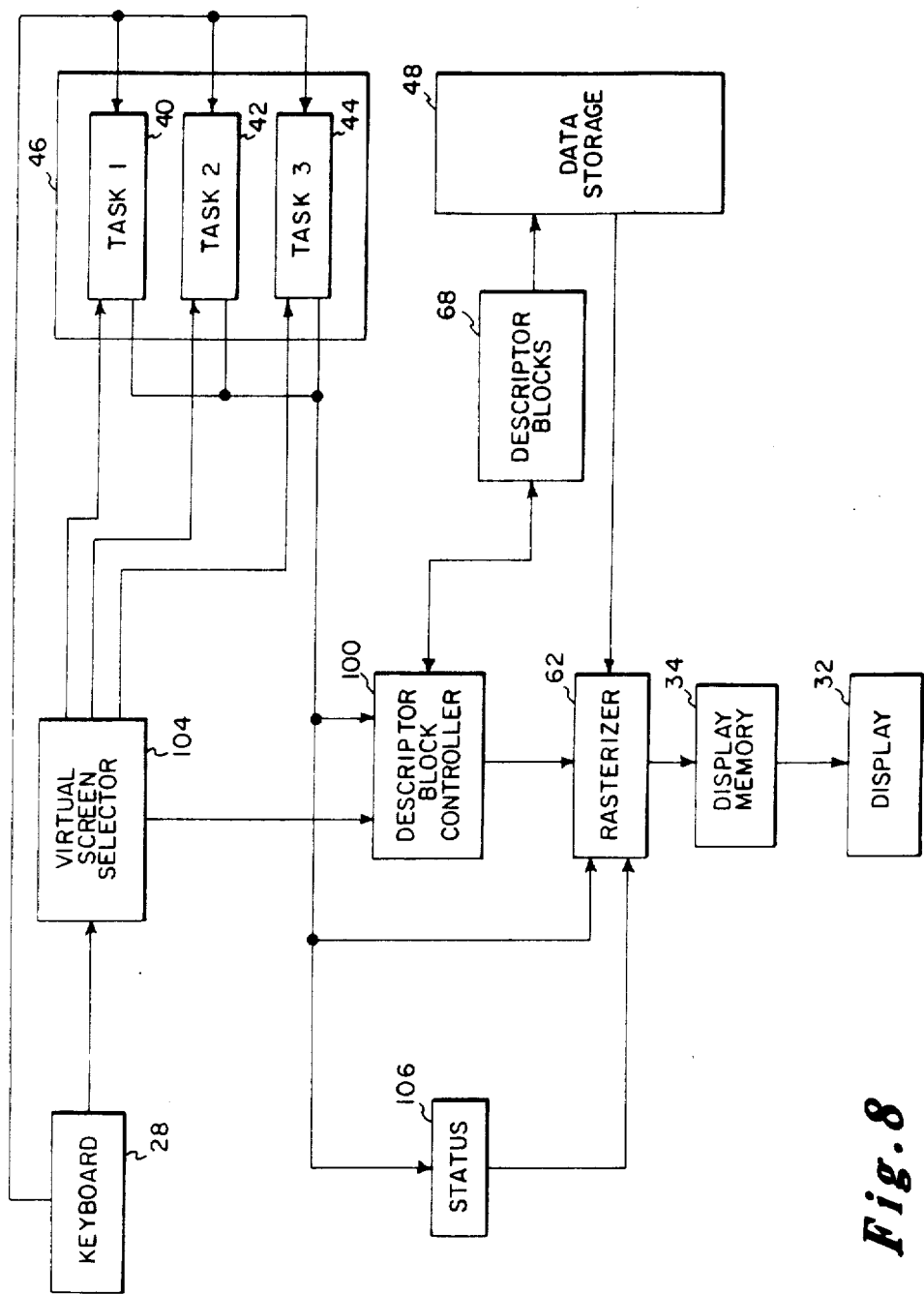


Fig. 8

1

SCREEN MANAGER FOR DATA PROCESSING SYSTEM

CROSS REFERENCE TO RELATED APPLICATIONS

This application is related to the following commonly assigned copending U.S. patent applications, namely: U.S. patent application Ser. Nos. 658,951 658,952, 658,953, 659,192 and 659,203 all filed on Oct. 9, 1984.

TECHNICAL FIELD

The present invention relates to display management in a data processing system and has particular application to word processing and office automation systems.

BACKGROUND

Word processing and office systems are primarily concerned with the generation, editing, displaying, printing and filing of documents. Those documents may include text in the case of word processing and they may include alphanumeric tables and graphics.

In addition to word processing tasks, the data processing systems may perform other tasks such as merging of forms, checking spelling through a dictionary task, spread sheet manipulation and communications. Less sophisticated systems allow only one such task to be performed at a given time. However, when a task requires little user input the keyboard remains idle. More sophisticated systems allow for multi-tasking. In such systems, an application task which requires little or no user input is performed by the system in a background mode; that is, the task does not interact with the keyboard and leaves the keyboard available to other tasks. A foreground task, on the other hand, which does require user input, interacts with the keyboard.

A common display technique for multi-tasking systems is referred to as windowing. In that technique, a document or a portion of a document being processed by the foreground task is predominantly displayed on the system display. Background documents relating to the background tasks are displayed in part so as to be perceived as being positioned below the foreground document but in partial view of the user. A background document can be moved into the foreground by positioning the display cursor over the selected background document. Only the task associated with the foreground document has access to the keyboard.

In another form of windowing, displays of documents associated with the various tasks are not overlapped. Rather, the various task windows are positioned in a side-by-side relationship.

DISCLOSURE OF THE INVENTION

The present invention relates to a data processing system having a central processing unit (CPU) which is controlled through an operating system program and application tasks software. Preferably, both the operating system and the application tasks are in the form of software which is loaded into a memory associated with the CPU. This system is also provided with a video display.

In accordance with one aspect of the invention, the CPU is able to process multiple application tasks together. A screen manager in the operating system is responsive to a plurality of application tasks to designate a plurality of virtual screens, all corresponding to the same single portion of the physical display screen.

4,688,167

2

The screen manager is also responsive to an input to the data processing system, such as a keyboard input, to select one of the virtual screens for display at the single portion of the physical display screen under control of an application task. Further, the screen manager controls display of identifiers at a second portion of the physical display screen. The identifiers correspond to the several virtual screens. Each identifier displayed in the second portion of the physical display may include an indication as to when an error exists in a particular background application task.

Preferably, the virtual screens are identified by descriptor data blocks stored by the screen manager. The descriptor data blocks designate portions of stored documents which are more directly handled by the application tasks and which are to be displayed. In response to requests by application tasks, the descriptor blocks are modified even when the virtual screens to which the blocks relate are held in background and thus not displayed.

The only task with which the keyboard interacts is that which controls the displayed virtual screen. A virtual screen associated with any task may be selected by a keystroke on the keyboard. The screen manager responds to the keystroke to move the selected virtual screen into foreground.

A memory associated with the display may be a bit map memory which includes individual data corresponding to each pixel of the display. A screen manager system within the operating system may include a software based rasterizer which generates the individual pixel data.

In accordance with another aspect of the invention, the operating system screen manager is responsive to an application task to designate as a plurality of viewports distinct portions of the physical display screen. Distinct sections of document data stored in memory under control of the applications task are designated for each viewport. Each viewport designated after a first viewport is formed as a subdivision of a larger viewport. The screen manager controls display of each designated section of data in its corresponding viewport portion of the physical display screen. The screen manager responds to the application task to change the designated viewport portions of the physical display screen and thus change the size, position and number of viewports. Also, the screen manager responds to the application task to independently change the logical position of a viewport with respect to the document files and to thus independently change the display of data in each viewport. The display may also be updated, through the screen manager, to include changes in the stored data made by the application task.

Preferably, the designated viewports of the physical display screen are stored in descriptor data blocks controlled by the screen manager. The descriptor blocks also point to the sections of document data stored in memory which are to be displayed in the viewports. The viewports may be independently associated with different documents and may be updated independently. With the size of each viewport indicated by a descriptor block, the operating system screen manager causes as much of a stored document designated by the descriptor block to be displayed as can be displayed in the particular viewport. With changes in the size of a viewport, under control of an application task, the screen manager automatically responds to increase or

4,688,167

3

4

decrease the amount of data from a stored document which is displayed. Also, changes in the logical position of a viewport relative to the stored document are controlled by the application task. For example, in response to an indication that the cursor has been moved beyond the logical position of the viewport, the screen manager may change the logical position of the viewport and automatically select the section of the document which is to be displayed. Under control of an application task, the screen manager may modify viewport descriptor blocks to further subdivide viewports or to merge viewports.

The viewport technique provides a flexible mechanism by which an application task can display data, most likely taken from different pages in the document files, in a side-by-side relationship. The ability to establish viewports is available to each application task. An application task can itself provide even greater flexibility by allowing for a subdivision of the sections of data, such as pages, which may be displayed in the viewports. Those subdivided areas can be independently controlled by the application task software but, unless modified by an application task, are seen as fixed side-by-side areas by the screen manager. Even further flexibility in the system is obtained by allowing each area to include multiple levels with one type of level including text and another including graphic information and the like. Those levels can be superimposed over each other when displayed in each area.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the invention will be apparent from the following more particular description of a preferred embodiment of the invention, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention.

FIG. 1 is a block diagram illustrating a work station embodying the present invention;

FIG. 2 illustrates a physical display screen displaying data in the system of FIG. 1 in accordance with principles of the present invention;

FIG. 3 is a schematic illustration of the logical arrangement of virtual screens to be displayed on the physical screen of FIG. 2;

FIG. 4 is a block diagram representation of the logical breakdown of the operating system of the work station of FIG. 1;

FIG. 5 illustrates the data structures in the system of FIG. 1;

FIGS. 6A and 6B respectively illustrate a physical display screen displaying in a single viewport three areas of text from a stored document and the logical position of the viewport with respect to the stored document;

FIG. 7A illustrates the physical screen of FIG. 6A after the single viewport has been subdivided into two viewports, and FIGS. 7B and 7C illustrate the logical location of each viewport over associated stored documents.

FIG. 8 is a functional block diagram of the system.

DESCRIPTION OF A PREFERRED EMBODIMENT

FIG. 1 illustrates a typical multi-task work station which, under proper software control, embodies the present invention. At the heart of the system is a central processing unit (CPU) 22 which is preferably a single chip microprocessor. The CPU is joined through a work station bus 24 to a high speed electronic memory 26 and peripheral devices. The peripherals include a keyboard 28, a magnetic disc storage unit 30, a display 32 which is preferably a cathode ray tube display and an associated display memory 34. At least one input/output unit 36 is also connected to the bus 24. The input/output unit 36 includes a communications port for communicating with a printer, other work stations or a main processing unit. Although the present invention is described with respect to a standalone word processing and office automation system, the invention is equally applicable to other systems such as distributed systems.

During start-up of this system, the operating software 38 is loaded into the memory 26 from the disc storage 30. That software, the operating system, controls the general operation of the CPU and the associated peripherals and serves as an interface between the CPU and peripherals and the applications software. Once the system is running under control of the operating system, the system user may select, through the keyboard 28, any of a number of application software packages from disc storage 30 and load them into the memory 26. In the illustration of FIG. 1, three independent application packages 40, 42 and 44 have been loaded into the memory at 46.

The user may also select, through the keyboard 28 and by way of the operating system 38, documents from the disc storage 30 to be stored at 48 in the memory 26. In the case of word processing, a document may correspond to pages of hard copy text which may be printed out directly through the I/O port 36 and a printer. A document may also include graphics data. On the other hand, the document may only include data which is intended to be processed and not printed directly onto hard copy. Thus, the term document merely applies to a unit of data to be processed by the CPU under control of one or more application tasks.

The system of FIG. 1 is a multi-tasking system. That is, the CPU is able to process several application tasks together in a multiplexed fashion. However, as will be described in greater detail below, the system user interacts with only one of those tasks at a time through the display 32 and keyboard 28. For that one task, which is the foreground task, the user may enter text data and text/document manipulation commands by means of keystrokes through the keyboard 28. The work station responds by executing in the CPU 22 the appropriate routines selected by the operating system 38 and, through the operating system, by the applications task 46. In executing those routines, the CPU may modify the contents of documents in the document files 48 and display results of the user input through the display 32.

A typical display on the physical display screen of display 32 is illustrated in FIG. 2. A display from the foreground task is provided on a major portion of the physical display screen indicated as the task screen 50. Under control of the operating system to be described below, the display on the task screen may be divided into a number of display viewports each of which independently displays a different set of information. The

4,688,167

5

viewports are shown separated by broken lines but such lines need not actually be displayed. As examples, on the display of FIG. 2 viewports are provided to display the document name, prompts, word processing page format, text, graphics, a user menu, and error messages. Of course, many other types of information can be displayed in different viewports and all of the viewports shown in FIG. 2 need not be displayed at any one time.

The viewport technique gives each application task great flexibility in designating the data to be displayed. That flexibility is obtained with little added complication to the application task software because it is controlled by the operating system once established by the application task. Once the viewports have been established, the application software need only be concerned with completing the task and modifying the data to be displayed as required by the task.

As noted above, only the task with which the user is interacting at any time is permitted to control a display on the physical screen. Although background tasks are not permitted to control the display 32, the operating system establishes virtual screens corresponding to background applications. Those virtual screens can be considered, as shown in FIG. 3, to be the logical equivalent of a stack of pages including virtual screens VS1, VS2, VS3 and VS4. Only one of those virtual screens, in this case VS3, is displayed on the physical screen. The other virtual screens are held by the operating system for display when called by the operator.

In order that the system user can always be aware of the status of virtual screens which are not displayed, the operating system provides virtual screen identifiers in an OS screen portion 52. Each identifier may name the virtual screen and may also provide an indication as to the status of the task. The OS screen 52 may also include a calendar and clock display.

The virtual screen approach to windowing provides the advantages of more conventional windowing techniques while avoiding many of the disadvantages of those techniques. The technique allows the screen manager to maintain an identification of a block of data to be displayed for each task being handled by the system, and the displays associated with the various tasks are readily identified by the user and moved to the foreground. With conventional windowing techniques the area of the physical display screen available to each task is substantially reduced. As a result, a lesser amount of information can be displayed for each task or the information must be reduced in size. With the present technique, the foreground virtual screen is displayed across virtually the entire physical display screen. Further, the software required to implement the technique can be much less complex. Only one virtual screen is displayed at a time, so it is not necessary to determine which areas of a background screen are covered by a foreground screen and which portions must thus be suppressed from the display. The resultant reduced complexity of the software allows for much faster operation.

It can be seen that the present system offers windowing at two levels. At a task level, in virtual screen windowing a task window covers virtually the entire physical screen. Within each virtual screen established by a particular application task, that task can subdivide the virtual screen into viewport windows. Because each viewport is associated with an active task, the viewports are positioned side-by-side.

FIG. 4 illustrates a logical breakdown of the operating system 38. Only those portions of the operating

6

system which primarily relate to the handling of peripherals, and in particular the display 32, are broken out in FIG. 4. The file management system 54 manipulates data to and from the keyboard, disc storage and input/output unit. The file management system interfaces with the peripherals through drivers 56 which include the software required for interfacing with the specific peripherals used. Of key importance with respect to this invention is the keyboard management driver 58.

The subject of the present invention is the screen manager system 60. The screen manager directly controls the information to be displayed in the operating system screen 52 (FIG. 2), and it interacts with the application tasks to determine the information to be displayed on the task screen and to define the virtual screens in background. The screen manager includes rasterizer software 62 which serves the function of a character generator and graphics generator for determining each pixel stored in the display memory 34 based on data received from document files 48.

The screen manager may also create a display not included in the data taken from the document files. For example, lines identifying the borders between viewports, such as illustrated by the broken lines in FIG. 2, may be created by the screen manager. Other borders, such as cross hatched strips, may also be created by the screen manager.

In addition to the memory in which the operating system programs are stored, additional memory is available to the operating system. A portion 64 of that memory is designated to carry specific pointers and status words. Of particular interest with respect to the present invention is the indication 66 of the number of the task which has control of the display and the keyboard at a given time. Indication 66 is set by the screen manager as it moves a virtual screen to the foreground. The keyboard management system 58 relies on that indication to determine which application task is to receive keyboard inputs.

The operating system also controls a larger section of memory 68. Virtual screen and viewport descriptor blocks, to be described in detail below, are stored in this section.

In order to display information on the display screen, an application task must first request that the operating system establish a virtual screen for the display. The CREATE VIRTUAL SCREEN request includes the task number of the requesting task which would have been previously designated by the operating system, a pointer to a six character string which identifies the virtual screen and a pointer to a page descriptor in the document files. It is the four character string which is displayed by the screen manager in the OS screen portion 52. In response to the request, the screen manager 60 creates a virtual screen descriptor block such as block 70 and a first viewport descriptor block such as block 77 and assigns a virtual screen number to the virtual screen block. The assigned virtual screen number is returned to the requester.

The data structures of the virtual screen and viewport descriptor blocks are shown in FIG. 5. A virtual screen description block includes a pointer to a first viewport descriptor block. The location and size of that first viewport corresponds to that of the entire virtual screen. If the primary viewport is subdivided into other viewports, each descriptor block of a subdivided viewport points to the next in a series of viewport descriptor blocks linked by pointers. Each descriptor block in-

4,688,167

7

cludes the location and size of its respective subdivision. It is also given a viewport number by which it can be identified in requests from the application task. Each viewport descriptor block in the chain points to a page descriptor block in the document files. As also shown in FIG. 5 each page descriptor block defines the size of the page and indicates the position of a cursor within the page.

The page descriptor to which the CREATE VIRTUAL SCREEN request and the resultant viewport point is a descriptor block defining the page of information which is to be displayed in the virtual screen. The virtual screen may be smaller than the page and in that case, the virtual screen can be considered a logical window over the upper left corner of the page when the virtual screen is first established. The cursor is also initially positioned in the upper left corner. The virtual screen window can be moved by movement of the cursor, indicated to the operating system by the applications task, or by a specific request from the applications task. When a cursor moves outside of that logical window on the page to be displayed, the screen manager automatically changes the position specification in the descriptor block. Thus, the descriptor block is dynamically controlled by the application task and at all times defines a logical window through which the cursor is viewed.

As noted above, the virtual screen or any viewport can be subdivided by establishing a new viewport descriptor block in the operating system. To that end, an application task issues a CREATE VIEWPORT request to the screen manager. That request includes the number of the virtual screen or any viewport to be subdivided, an indication as to whether the subdivision is to be horizontal or vertical, an indication as to whether the subdivision is to be fixed or proportional, an indication of the size of the subdivision. In response to such requests, the operating system establishes a descriptor block such as descriptor block 72 in memory and references that descriptor block in the descriptor block of a prior viewport which is being subdivided. Subsequently, by means of an ASSIGN request the application task can provide the screen manager with a pointer to be included in the descriptor block. That pointer points to the page descriptor of data which is to be displayed in the new viewport. Again, the viewport window is logically positioned at the upper left corner of the page. Also, the cursor is initially positioned at the upper left corner of the viewport. As with the virtual screen, each viewport within the screen can be repositioned with respect to its respective page in the document files by means of cursor movement indicated to the screen manager by the application task or by specific requests from the application task.

It can be seen that any number of virtual screens can be established by the screen manager in response to requests from application tasks and each virtual screen can be subdivided into any number of viewports by additional requests from the respective application task. Each virtual screen and each viewport is defined by a descriptor block which sets the size of the virtual screen or viewport, points to a page or document in the document files which is to be displayed in the virtual screen or viewport and sets the logical position of the screen or viewport relative to the page or document.

When a virtual screen is in the foreground, the screen manager relies on the descriptor blocks to designate the data from the document files which is to be displayed.

8

That data is passed through the rasterizer 62 of the screen manager to generate the signal to be applied at each pixel of the display screen. The code for each pixel is stored in an 800 by 300 bit display memory 34. The screen manager also selects the information to be displayed on the operating system screen 52 designated in a descriptor block 74 and, through the rasterizer 62, stores corresponding pixel information in the memory 34.

The data stored in the memory 34 is continuously displayed by the display 32 until the memory is updated by the screen manager. The display memory 34 is updated in either of two situations. Where a foreground virtual screen or viewport descriptor block is modified, as when the logical position of a viewport on a page in the document files is changed, the screen manager immediately updates the display memory to pass the freshly indicated data from the document files 48 through the rasterizer 62. On the other hand, the application task may continuously update the data in the document files. The screen manager is unaware of those changes until an UPDATE request is made by the application task and does not update memory 34 until such a request is received. In response to the UPDATE request, the screen manager again selects the data from the document files to which the descriptor blocks point and passes that data through the rasterizer to update the display memory 34.

The screen manager is not concerned with the data included in document files pointed to by descriptor blocks associated with background virtual screens. The screen manager only becomes concerned with that information when the information is pointed to by a foreground descriptor block and memory 34 is to be updated. At that time the information is passed through the rasterizer to the display memory. Therefore, the screen manager does not respond to any update request with respect to background virtual screens. On the other hand, the virtual screen and corresponding viewport descriptor blocks must at all times be up to date so that when a particular background virtual screen is selected for movement into foreground, the information that the application task requires to be displayed is immediately and properly displayed. Therefore, the screen manager must respond to specific requests to modify background descriptor blocks and to cursor movements which move the logical positions of background descriptor blocks even though modifications of background descriptor blocks do not result in an immediate response on the display 32.

In order to minimize the amount of data which must be updated, the application task requesting an update may specify less than an entire virtual screen or viewport. An example is illustrated in FIGS. 6A and 6B. FIG. 6A represents the physical screen which displays a virtual screen which has not been subdivided into viewports. The logical position of the virtual screen 76 over a page 78 in the document files is illustrated in FIG. 6B. The page 78 is divided in the document files into three areas A, B and C. Areas A and B may, for example, correspond to two columns of text and area C may correspond to text extending across the full width of the page. In a particular application, it may only be necessary to update area B on the page 78. Thus the application task requests the operating system screen manager to update area B only. The screen manager recognizes that only the portion of area B overlapping the virtual screen 76 need be updated in the display

4,688,167

9

memory 34. Therefore, only the cross-hatched area in FIG. 6B is actually updated. By thus limiting the amount of information which must be updated, the updating function can be completed in less time.

The subdivision of each page into areas is accomplished by the data structures of FIG. 5. It can be seen that the page descriptor block includes a pointer to an area descriptor block. The area descriptor block establishes the locations of diagonal corners of a square area. It may also include indications of the left and right margins on which the screen manager may rely to minimize the amount of rasterization processing required for the area. The area also points to one or both of a text column descriptor block and a layer descriptor block. The column descriptor block includes a number of pointers to several lines of text included in the column. Each line descriptor block to which the column block points includes one or more strings of text. The same area may also point to a layer descriptor block which in turn points to either a graphics descriptor or an image descriptor. Because the area block can point to both a text column block and a layer block, text, graphics and imagery can be superimposed in the single area.

It can be noted that the area descriptor block also includes a pointer to the next area within the page. That area may similarly point to text and/or graphics data and a subsequent area.

FIGS. 7A through 7C illustrate the subdivision of the virtual screen 76 of FIG. 6B into two viewports. A CREATE VIEWPORT request is first made to the screen manager. That request defines the size and location of a viewport shown at 80 to the right of the physical screen in FIG. 7A. The two viewports are shown separated by broken lines, but such lines need not be included on the actual display. With the establishment of the viewport 80, the screen manager automatically reduces the text from the document files which is displayed in the primary viewport 76, as can be seen by comparison of FIGS. 6B and 7B.

An ASSIGN request is next issued by the application task to the screen manager to assign the viewport 80 to a page of data in the document files 48. That page 82 may, for example, include graphics 83. The newly assigned viewport is initially positioned in the upper left corner of the page 82. Only that part of the page logically within the viewport, as illustrated in FIG. 7C, is actually displayed, as shown in FIG. 7A. The viewport can be logically repositioned on the page by cursor movement or specific command.

To subsequently remove the viewport 80, a MERGE request is made by the application task to merge the viewport 80 back into the virtual screen. The result is that the primary viewport 76 returns to the full size of the virtual screen as shown in FIGS. 6A and 6B.

It can be understood that the virtual screens are created by the application task and are resources assigned to the application task. An application task may create multiple virtual screens. The application task must release the virtual screen before it terminates or whenever it does not need the virtual screen. The virtual screen is released by a DELETE request from the application task to the screen manager. The screen manager then deletes the virtual screen and the corresponding viewport descriptor blocks from the data structures and updates the display to show the next sequential virtual screen.

A functional block diagram of the system is shown in FIG. 8. Through a controller 100 in the operating sys-

10

tem, each application task 40, 42 and 44 is able to create and modify virtual screen and viewport descriptor blocks 68 (FIGS. 4 and 5). This controller handles the several functions described above. In particular, it implements the CREATE VIRTUAL SCREEN, CREATE VIEWPORT, ASSIGN, UPDATE, MERGE, AND DELETE functions with respect to particular descriptor data blocks 68.

A virtual screen selector 104 responds to keyboard input to designate the application task which has access to the keyboard and to indicate that task to the descriptor block controller 100. The controller 100 in turn selects the virtual screen descriptor blocks associated with the selected application task. The selected viewport descriptor blocks are used by the controller 100 to designate viewports within the virtual screen to the rasterizer 62. The selected descriptor blocks also designate the data in storage 48 which is also to be applied to the rasterizer 62.

Finally, the status of each application task is monitored by monitor 106 and applied to the rasterizer 62. From these inputs, the rasterizer generates a complete video display. Updating the display may be made in response to signals from application tasks when the underlying data is changed or in response to changes in descriptor blocks.

The present system has several advantages over conventional windowing techniques in multi-tasking systems. In conventional windowing techniques, several displays corresponding to the virtual screens of the present application are overlapped but spatially offset from each other on the physical screen. The result is the need for a very complex rasterization routine. To handle that routine rapidly, it is best handled by hardware rather than by software control. However, hardware control is relatively inflexible, particularly with respect to type of character which is displayed. By displaying only one virtual screen at a time, the rasterization process is greatly simplified and can be handled rapidly under software control. With software control, much greater flexibility is obtained.

The present technique also allows for the virtual screen of primary interest to make up a much larger portion of the physical screen. The use of the operating screen 52 in the display gives the operating system sufficient opportunity to keep the user informed as to the status of virtual screens which are not displayed.

Further, the ability of the operating system to establish viewports in each virtual screen greatly adds to the flexibility of the system, particularly with respect to displaying different types of data such as text and graphics. The information displayed in different viewports can also be selected from different pages and even different documents in the document files 48. The example of displaying text adjacent to graphics using the viewport technique has previously been noted. Establishing viewport descriptor blocks for other items such as the menu and error messages of FIG. 2 makes the screen manager operations extremely flexible. It also minimizes the amount of updating of the screen. For example, in order to update the prompts viewport, which may require frequent updating, it is not necessary to as frequently update the entire screen. Similarly, when word processing, it may only be necessary to update the text viewport and not the other viewports at particular stages of an application task.

The ability of the applications task to further subdivide pages into areas adds yet another dimension to the

4,688,167

11

control of information to be displayed. It allows the application task to establish areas to be displayed in a relatively fixed relationship as far as the screen manager is concerned; whereas, the viewport technique requires the screen manager to handle each viewport more independently. Establishing areas simplifies certain tasks of the application software such as formatting, wrap-around within columns and the like.

Finally, the ability to superimpose text and graphics for imagery adds yet another dimension to the display of information.

While the invention has been particularly shown and described with reference to a preferred embodiment thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention as defined by the appended claims.

I claim:

1. In a data processing system comprising a CPU controlled through an operating system and application tasks so as to be able to process multiple application tasks together, a data storage memory, a keyboard for interacting with application tasks, and a video display, having a physical display screen, responsive to the application tasks, the operating system having a screen manager comprising:

means responsive to a plurality of application tasks to designate a plurality of virtual screens, all virtual screens corresponding to the same, single portion of the physical display screen;

means responsive to an input to the data processing system to select one of the virtual screens for display at said single portion of the physical display screen under control of an application task; and

means for controlling display, at a second portion of the physical display screen, of identifiers corresponding to the designated virtual screens.

2. A data processing system as claimed in claim 1 wherein the virtual screens are designated in descriptor data blocks stored by the screen manager and the descriptor blocks point to stored data processed through the application task which is to be displayed, and wherein the screen manager further comprises means for modifying the descriptor blocks in response to application tasks regardless of whether a virtual screen associated with a particular descriptor block is at that time being displayed.

3. A data processing system as claimed in claim 1 wherein the only task with which the keyboard interacts is that which controls the displayed virtual screen.

4. A data processing system as claimed in claim 3 wherein the screen manager responds to a keystroke to update the display to a different virtual screen.

5. A data processing system as claimed in claim 4 wherein the identifiers displayed in the second portion of the physical display screen indicate the status of the background tasks.

6. A data processing system as claimed in claim 1 wherein the information to be displayed is stored in a display memory on a pixel by pixel basis and the screen manager further comprises a software based rasterizer for generating individual pixel data from data in the data storage memory pointed to by the designated virtual screens.

7. In a data processing system comprising a CPU controlled through an operating system and an application task so as to be able to process the application task, a data storage memory, and a video display, having a

12

physical display screen, responsive to the application task, the operating system having a screen manager comprising:

means responsive to an application task to designate as a plurality of viewports distinct portions of the physical display screen and to designate corresponding distinct sections of data, stored in the data storage memory, to be displayed in the respective viewports, each viewport designated after a first viewport being formed as a subdivision of a larger viewport;

means for controlling display of each designated section of data in its corresponding viewport portion of the physical display screen;

means responsive to the application task for changing the designated distinct viewport portions of the physical display screen and for independently, for each viewport, changing the designated distinct sections of stored data corresponding to each viewport and to thus change the display of data; and

means for updating the display to include changes in the stored data made by the application task.

8. A data processing system as claimed in claim 7 wherein each of the viewports and the corresponding sections of data are designated in descriptor data blocks by the screen manager.

9. A data processing system as claimed in claim 8 wherein each descriptor block includes the size of a viewport and its logical location relative to data in the data storage memory.

10. A data processing system as claimed in claim 9 wherein the screen manager further comprises means for changing the logical position specified in the viewport descriptor data block in response to a cursor position indicated by the application task.

11. A data processing system as claimed in claim 8 wherein the screen manager comprises means for further subdividing viewports into smaller viewports.

12. A data processing system as claimed in claim 11 wherein the screen manager comprises means for merging viewports to remove a subdivision of a viewport.

13. A data processing system as claimed in claim 7 wherein the distinct sections of data may be included in distinct documents in the data storage memory.

14. A data processing system as claimed in claim 7 wherein the means for updating the display to include changes in the stored data made by the application task provides for updating of viewports independently.

15. A data processing system as claimed in claim 7 further comprising in application task software means for subdividing said distinct sections of data, stored in the data storage memory, into areas displayed in the fixed side-by-side relationship when displayed through the screen manager and independently controlled by the application task software.

16. A data processing system as claimed in claim 15 wherein the application task software includes means for designating both text and other data to be superimposed in the area.

17. A data processing system as claimed in claim 7 wherein the screen manager comprises means for providing a border display at the interfaces of viewports.

18. In a data processing system comprising a CPU controlled through an operating system and application task software so as to be able to process the application task, a data storage memory, and a video display, having a physical display screen, responsive to the application task, the system further comprising

4,688,167

13

A. a screen manager in the operating system comprising:

- means responsive to an application task to designate as a plurality of viewports distinct portions of the physical display screen and to designate corresponding distinct sections of data, stored in the data storage memory, to be displayed in the respective viewports;
- means for controlling display of each designated section of data in its corresponding viewport portion of the physical display screen;
- means responsive to the application task for changing the designated distinct viewport portions of the physical display screen and for independently, for each viewport, changing the designated distinct sections of stored data corresponding to each viewport and to thus change the display of data; and
- means for updating the display to include changes in the stored data made by the application task; and

B. the application task software comprising means for subdividing said distinct sections of data, stored in the data storage memory, into areas displayed in a fixed side-by-side relationship when displayed through the operating system and independently controlled by the application task software.

19. A data processing system as claimed in claim 18 wherein the application task software includes means for designating both text and other data to be superimposed in the area.

20. In a data processing system comprising a CPU controlled through an operating system and application tasks so as to be able to process multiple application tasks together, a data storage memory, a keyboard for interacting with application tasks, and a video display, having a physical display screen, responsive to the application tasks, the operating system having a screen manager comprising:

- means responsive to a plurality of application tasks to designate a plurality of virtual screens for independent display on the physical display screen;
- means responsive to an input to the data processing system to select one of the virtual screens for display under control of an application task;
- means responsive to an application task to designate as a plurality of viewports distinct portions of the physical display screen and to designate corresponding distinct sections of data stored in data storage memory to be displayed in the respective viewports;
- means responsive to an application task for changing the designated distinct viewport portions of the physical display screen and for independently, for each viewport, changing the designated distinct sections of stored data corresponding to that viewport and to thus change the display of data; and
- means for updating the display to include changes in the stored data made by an application task.

21. A data processing system as claimed in claim 20 wherein each of the viewports and the corresponding sections of data are designated in descriptor data blocks by the operating system.

22. A data processing system as claimed in claim 21 wherein the virtual screens are designated in descriptor data blocks stored by the operating system and the descriptor blocks point to stored data processed through the application task which is to be displayed,

14

and wherein the screen manager further comprises means for modifying the descriptor blocks in response to application tasks regardless of whether a virtual screen associated with a particular descriptor block is at that time being displayed.

23. A data processing system as claimed in claim 21 wherein the only task with which the keyboard interacts is that which controls the displayed virtual screen.

24. A data processing system as claimed in claim 20 wherein the information to be displayed is stored in a display memory on a pixel by pixel basis and the screen manager further comprises a software based rasterizer for generating individual pixel data from data in the data storage memory pointed to by the designated virtual screens.

25. A data processing system as claimed in claim 20 further comprising means for controlling display, at a second portion of the physical display screen, of identifiers corresponding to the designated virtual screens.

26. A data processing system as claimed in claim 20 wherein the operating system comprises means for providing a border display at the interfaces of viewports.

27. A data processing system as claimed in claim 20 further comprising in application task software means for subdividing said distinct sections of data, stored in the data storage memory, into areas displayed in the fixed side-by-side relationship when displayed through the screen manager and independently controlled by the application task software.

28. A data processing system as claimed in claim 27 wherein the application task software includes means for designating both text and other data to be superimposed in the area.

29. A method of displaying data in a data processing system comprising:

- responsive to a plurality of application tasks, designating in an operating system of the data processing system a plurality of virtual screens, each virtual screen corresponding to a portion of the physical display screen;
- selecting one of the virtual screens and displaying that virtual screen in a first portion of the physical display screen in response to an input to the data processing system; and
- displaying in a second portion of the physical display screen identifiers corresponding to the designated virtual screens.

30. A method as claimed in claim 29 wherein the virtual screens are designated in descriptor data blocks stored by the operating system and the descriptor data blocks point to stored data processed through the application task which is to be displayed, the method further comprising modifying the descriptor blocks in response to application tasks regardless of whether a virtual screen associated with a particular descriptor block is at that time being displayed.

31. The method as claimed in claim 29 further comprising generating in a software based rasterizer individual pixel data from data in the data storage memory to which the designated virtual screens point and storing the individual pixel data in a display memory on a pixel by pixel basis.

32. A method of displaying data in a data processing system comprising:

- responsive to an application task, designating in an operating system as a plurality of viewports distinct portions of a physical display screen and designating corresponding distinct sections of data

15

stored in a data storage memory to be displayed in the respective viewports;
displaying each designated section of data in its corresponding viewport portion of the physical display screen;
responsive to the application task, changing the designated distinct viewport portions of the physical display screen and independently, for each viewport, changing the designated distinct sections of stored data corresponding to each viewport to thus change the display of data; and
updating the display to include changes in the stored data made by the application task.
33. A method as claimed in claim 32 wherein each of the viewports in the corresponding sections of data are designated in descriptor data blocks by the operating system of the data processing system.
34. A method as claimed in claim 33 wherein each descriptor block includes the size of the viewport and its logical location relative to data in the data storage memory, the method further comprising changing the logical location specified in the viewport descriptor data block in response to a cursor position indicated by the application task.
35. A method as claimed in claim 32 further comprising subdividing said distinct sections of data, stored in the data storage memory, into areas displayed in a fixed side-by-side relationship when displayed through the operating system and independently controlled by the application task software.
36. A method as claimed in claim 35 further comprising dividing for an area text data and other data to be superimposed in display of the area.
37. A method of displaying data in a data processing system comprising:
responsive to a plurality of application tasks, designating in an operating system a plurality of virtual screens for independent display on a physical display screen;
selecting one of the virtual screens and displaying that virtual screen in a first portion of the physical display screen in response to an input to the data processing system;
responsive to an application task, designating as a plurality of viewports distinct portions of the phys-

16

ical display screen and designating corresponding distinct sections of data stored in data storage memory to be displayed in the respective viewports;
responsive to an application task, changing the designated distinct viewport portions of the physical display screen and independently for each viewport changing the designated distinct sections of stored data corresponding to that viewport to thus change the display of data; and
updating the display to include changes in the stored data made by an application task.
38. A method as claimed in claim 37 wherein each of the viewports in the corresponding sections of data are designated in descriptor data blocks by the operating system of the data processing system.
39. A method as in claim 38 wherein the virtual screens are designated in descriptor data blocks stored by the operating system and the descriptor data blocks point to stored data processed through the application task which is to be displayed, the method further comprising modifying the descriptor blocks in response to application tasks regardless of whether a virtual screen associated with a particular descriptor block is at that time being displayed.
40. A method as in claim 37 further comprising generating in a software based rasterizer individual pixel data from data in the data storage memory to which the designated virtual screens point and storing the individual pixel data in a display memory on a pixel by pixel basis.
41. A method as in claim 37 further comprising displaying, at a second portion of the physical display screen, identifiers corresponding to the designated virtual screens.
42. A method as claimed in claim 37 further comprising subdividing said distinct sections of data, stored in the data storage memory, into areas displayed in a fixed side-by-side relationship when displayed through the operating system and independently controlled by the application task software.
43. A method as claimed in claim 42 further comprising dividing for an area text data and other data to be superimposed in display of the area.
* * * * *

50

55

60

65